EFFICIENTLY FINDING (NEARLY) MINIMAL FST OF REPETITIVE UNSEGMENTED DEMONSTRATION DATA

Frederick L. Crabbe
Computer Science Department, US Naval Academy, 572M Holloway Rd Stop 9F, Annapolis, Maryland 21402, U.S.A.

Keywords: Robots, Plan Learning, Learning from Demonstration, Minimal Programs.

Abstract: This paper presents an algorithm that enables a robot to learn from demonstration by inferring a nearly minimal *plan* instead of the more common *policy*. The algorithm uses only the demon- strated actions to build the plan, without relying on observation of the world states during the demonstration. By making assumptions about the format of the data, it can generate this plan in $O(n^5)$.

1 INTRODUCTION

Learning from Demonstration (LfD) is an established

eld in which robot controllers are devel- oped from example data. In most cases, this takes the form of learning a policy (Bertsekas, 2005) that maps world states directly to outputs (Argall et al., 2009) . There are however other possible models for LfD, such as inferring planning operators or even direct plans or programs. This paper examines the problem from a non-policy perspective, inferring a plan represented as a Finite State Transducer (FST) from example data that is repetitive (a set of actions is repeated, with small variations each repetition).

By making the assumption of repetitive input data, we enable the inference of a nearly mini- mal size program from the data (minimality tends to have good generalization properties (Blumer et al., 1987)). In the general case, inferring a min- imal program is very expensive, and inferring a minimal Finite State Automaton is NP-complete (Gold, 1978; Pitt and Warmuth, 1993). We claim that the repetitive nature of many robot tasks makes this input-as-repeating-data a reasonable assumption. This assumption, plus an assumption that the world state only needs to be sampled at certain times in the repetitions, lead to fast algorithms, inferring an FST in $O(n^5)$.

An example real-world repetitive task might be the process of loading a container ship. In it the loader must repeatedly: a) move container in position; if load has shifted, b) repack; c) lift con- tainer; d) move to location on ship; e) secure con- tainer; if there is void space, f) fill with dunnage; g) lock doors (McNamara, 2010). A robot at- tempting to infer this algorithm might see a long sequence of world states paired with actions. For example, if we number world states arbitrarily, and label actions as above, the observed sequence might be: (0/a, 8/c, 8/d, 9/e, 8/g, 1/a, 5/b, 7/c, 9/d, 9/e, 6/g, 2/a, 9/c, 8/d, 2/e, 9/f, 2/g, 3/a, 5/b, 0/c, 8/d, 9/e, 8/f, 2/g, 0/a, 5/c, 0/d, 9/e, 8/g). Our approach strips out the actions: (acde- gabcdegacdefgabcdefgacdeg) and uses this to de- termine the basic loop, adding the world state in- formation back in at the end, as shown in Figure 1. As discussed below, the agent using this pro- gram samples the world state at the start of each pass through the loop, and uses that information to determine all actions through that pass.

2 APPROACH

Because the general problem of inferring a min- imal finite state machine from a data set is difficult, we make some restrictive assumptions on the structure of the possible programs. We assume all programs consist of a repeated sequence of actions (no nested loops) where within individual passes of the loop, there might be small deviations from the typical pass. We will infer the machine from the observed behavior only, and apply the sensor observations of the demonstrator at the end of the process. Thus, the formal problem is, given a demonstration sequence consisting of world-state/action pairs, infer a minimal FST such that the action sequence would be generated by repeated applications of that FST. Following Veloso and Veeraraghavan (Veeraraghavan and Veloso, 2008), we assume that at the start of each loop the world-state contains sufficient information to determine the path through the FST. The robot makes the decision about the relevant properties of the envi-

L. Crabbe F..
EFFICIENTLY FINDING (NEARLY) MINIMAL FST OF REPETITIVE UNSEGMENTED DEMONSTRATION DATA.
DOI: 10.5220/0003881906730678
In *Proceedings of the 4th International Conference on Agents and Artificial Intelligence* (SSIR-2012), pages 673-678
ISBN: 978-989-8425-95-9
Copyright © 2012 SCITEPRESS (Science and Technology Publications, Lda.)

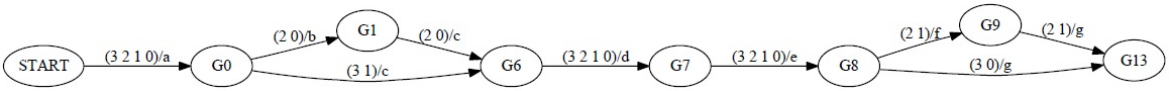

Figure 1: Contents of the loop in the program inferred in the loading example, represented as a Mealy Machine. Each transition is labeled with a set of world states in parentheses, a slash, and then actions to take given those world states. During execution, the world state is observed at the start, and that value is used to select transitions throughout that pass in the loop. Thus, if in world state 1 at the start, the robot would move from Start to G0, G6, G7, G8, G9, and finally G13, outputting (acdefg) for that pass through the loop. World states 5-9 are not used in the machine because they are never true at the beginning of a loop pass.

ronment at the start of the pass through the loop, following branches determined at that start (in an extreme case, imagine the robot consulting an information board at the start of a pass, and using that information to determine what actions to take through that pass). The ramifications of this will be discussed in section 5. For example, figure 2 shows the contents of the loop where the robot executes action a then b, and then if condition 0 is true, execute action c, or if condition 1 is true, execute action d.

Current state-of-the-art FSA inference algorithms use a set of short strings as input (Bugalho and Oliveira, 2005). The demonstration data for our problem is, in contrast, single long sequence of actions. Our approach is to first efficiently partition the action sequence using a technique for finding approximate repetitions in strings, and then use the set of substrings to generate the transducer. Of course, even given partitioned actions, the problem of generating the minimal transducer is, as noted above, NP-complete. However, the assumption that the input is known at the beginning of each pass through the loop provides us with an efficient algorithm to find the minimal FST.

2.1 Notation

As in the above example, we will represent an action in a sequence as a single character and a complete action sequence as a string. When a string has been partitioned into substrings, where each substring represents a pass through the loop, each substring will be labeled with a number, corresponding to the matching world conditions that must later be determined from the world state that matches the start of each subsequence.

3 ALGORITHM

The algorithm consists of computing one or more partitions of the action string, using the set of strings in the partition to generate a FST, and finally apply the world-state conditions from the original data to create the FST.

3.1 Generating Partitions

The partitioning problem can be stated as, given a string x of length n, generate a set of m substrings, $x[1...i_1], x[i_{1+1}...i_2], ..., x[i_m..n]$ that will be used to generate a minimal FSA. To do this, we rely on the assumption that most of the iterations through the single loop are similar if not identical, with some steps inserted or deleted. Under this assumption, the problem resembles finding an *approximate repetition* (Sim et al., 2001). In an approximate repetition problem, a string p is a *period* of x, if $x = p_1 p_2 ... p_m$, where each p_i is an edited version of p. Given an edit distance function $\delta(.,.)$, the problem becomes finding the best $p(p^*)$, s.t. given all possible partitions p_i,

$$p^* = \arg\min_p \max_i \delta(p, p_i)$$

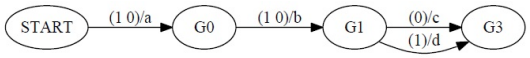

Figure 2: FST inferred from the action sequence "abcabd".

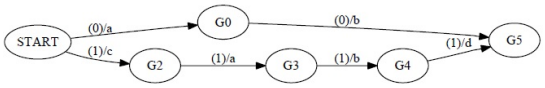

Figure 3: A minimal FST for the actions sequence "abcabd", when partitioned with the period "ab".

That is, assuming that x is made up of approximate repetitions of some string, the period p^* is the string for which there exists a partition of x where the distance between p^* and the substring it is furthest from is less than any other combination of p_i's and partitions. As we shall see, the process of finding p^* will also partition the string into its component p_i.

The approximate repetition problem described above is different from the sort of partitioning required for our problem in three ways. First, since we a measuring the size of the overall program, instead of the maximum distance from p^* we are looking for the sum of all the distances between p^* and each p_i. Fortunately, this is easily fixed, as described below. Second, the sum of the edit distances does

not match perfectly with the size the resulting FSA. For example, if $x =$ "*abcabd*", the p^* was "abc" and the total estimated program size is 4: 3 to generate the period "abc", plus 1 substitution for "abd" (figure 2). However, selecting the period as "ab" also yields an estimated program size of 4 (2 to generate the period, plus two edits to generate "cabd", yet this partition results in a larger FST than the former (figure 3). Thus the partition can under-estimate the size of the resulting program. In practice, the when the partition algorithm returns a bad partition, it also returns a good one with the same cost. The third difference between the approximate repetition problem and our partition problem is that the former looks at the pairwise cost between p^* and each p_i, whereas in the FST multiple p_i's can also interact. For example, given $x =$ "*abcabcabcabdabd*" and $p =$ "*abc*", the partition algorithm partitions to p_0 :"abc", p_1 :"abc", p_2 :"abc", p_3 :"abd", p_4 :"abd" and estimates a cost of 5 (3 for p_0, 0 for p_1 and p_2, one each for the replacing of "c" with "d"). Of course, since the "d" is the same for p_3 and p_4, the real cost should be 4. Thus the partition can also over-estimate. The implications of these two inaccuracies is discussed more is section 5.

If we allow p to be any string, the partitioning problem is known to be NP-complete (Sim et al., 2001), but if we assume that p occurs in x, then it can be found in polynomial time. We can modify the algorithm from Sim et al. (Sim et al., 2001) to find p^* for our problem, shown in algorithm 1.

Algorithm 1

$p_{max} = \emptyset$
$t_{max} = 0$

For each $p \in Substrings(x)$
1. *For each* $i | 0 \leq i \geq n$
   (a) Compute the edit distance table δ *from p to* $x[i..n]$
2. $t_0 = 0$
3. *For each* $i | 0 < i \leq n$
   (a) $t_i = min_{0 \leq h < i}(t_h + \delta(p, x[h+1..i]))$
4. *if* $t_n + |p| > t_{max}$, $t_{max} = t_n$, $p_{max} = p$
   return p_{max}

In the main loop (step 3) each t_i is the total edit distance using p as the period, up through position i. In step 3a, we consider the cost of partitioning from h to i, which would be the distance from p to $x[h+1..i]$, plus the cost of all of the other partitions, t_h.

The algorithm above not only finds the best p for the string x, if we remember h for each term selected by the min in step 3a, the resulting list of h's gives us the points to split the string x, thus generating our partitions for the next stage.

3.2 Building the FST

Once we have broken the action sequence into a set of subsequences, we can begin to find the minimal FST that generates those strings. One standard approach to this problem is to first build a tree known as the Augmented Prefix Tree Acceptor (APTA) (Bugalho and Oliveira, 2005), an FSA in tree form that generates each of the action substrings (figure 4). Once the APTA is built, there are several algorithms that attempt to minimize it by merging pairs of mergeable states (Lang et al., 1998). These algorithms search the space of possible merges to find the minimal FSA. Unfortunately, this approach also has exponential time complexity.

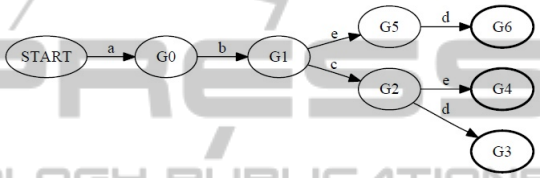

Figure 4: The APTA generated from the set of strings, {"*abcd*","*abce*","*abed*"}.

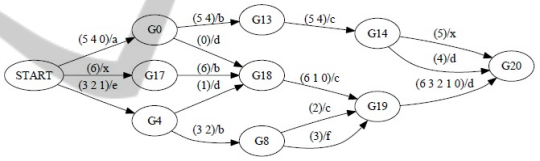

Figure 5: Final FST from the action sequence "adcdedcdebcdebfdabcdabcxxbcd".

If we include the assumption that the world- state information given to the robot at the beginning of each pass through the loop, then we can consider the world-state information to be the same for each or the steps in the subsequence. For example, if a world-state/action sequence was (1/a, 4/b, 2/c, 2/a, 3/b, 1/c) the action sequence would be (abcabd), which is partitioned into (abc) and (abd). We then relabel the actions with the world state at the start of each subsequence: (1/a, 1/b, 1/c) and (2/a, 2/b, 2/d). If we treat our APTA as a FSA, temporarily ignoring the world state, and label the transitions with just the actions, the result is a Deterministic Finite-state Automaton (DFA). This DFA can be minimized with known algorithms in $O(nlogn)$ time (Hopcroft, 1971). Once we have a minimal DFA, we can add the world-state labels to the transitions without harming the deterministic property of the automaton because of our assumption about the structure of the world state information. We can easily show that this results in an unambiguous FST:

Lemma 1. *Minimizing the APTA as a DFA, and then applying the world states from the starts of each subsequence to the entire subsequence yields a minimal unambiguous FST.*

Proof.

1. The original APTA contains no cycles, by definition of tree, therefore the minimized DFA contains no directed cycles, or it would accept arbitrarily long strings not accepted by the APTA.
2. When converting the DFA to an FST, the addition of constraints in the form of the world states (the numbers) on the transitions cannot enable a smaller machine. The FST could be made smaller than the DFA only if adding the world states enabled more merges on the minimal DFA. Since mergeability of two states is still determined in part by the output character, the additional constraints will not enable more states to merge in either the minimal or any other DFA. Therefore the FST is also minimal.
3. Non-determinism in the FST could only come from more than one transition from a node with the same world-state on each transition. Since each partition substring is uniquely labeled, ambiguity cannot arise from labels from multiple substrings. Because there are no directed cycles, when labeling the transitions for a particular substring, each node can be visited at most once. Therefore the FST is unambiguous.

Algorithm 2

1. Compute APTA from data, ignoring worldstate.
2. Minimize APTA as a deterministic finite state automaton.
3. For each data string d
(a) s =start state
(b) For each character c ∈ d
i. t = transition from s that matches c
ii. Label t with the world-state from d
iii. s = NextState(t)

3.3 Example

An example run through the algorithm follows. Given an initial action sequence of "adcdedcdebcdebfdabcdabcxxbcd", the partitioning portion generates two potential partitions, each with an estimated program size of 12. The first uses "abcd" as p, partitioning the string as, {"adcd" "edcd" "ebcd" "ebfd" "abcd" "abcx" "xbcd"}. The second partition uses "ebcd" as p, but ends up with the same partition. The FST builder takes that partition and generates the FST shown in figure 5.

3.4 Run Time Analysis

In algorithm 1, the main loop is executed $O(n^2)$ times. Building each distance table takes (letting $q = |p|$, and i is from line 1 (a) in algorithm 1) $O(qi)$, so that all the tables take $O(n^2q)$, roughly $O(n^3)$. In step 3a, there are n comparisons in each call to min, and t_i is computed n times, for a total of $O(n^2)$, therefore algorithm 1 is $O(n^5)$. In algorithm 2 step 1 is $O(n)$, step 2 can be done in $O(nlogn)$ (Hopcroft, 1971), and the loop in step 3 is just $O(n)$, making algorithm 2 $O(nlogn)$ and the overall process $O(n^5)$.

Once caveat to this analysis is that it assumes the action data genuinely comes from a loop and there is a sensible period. If there is no good period, such as when all the characters are unique, all partitions are equally bad. In this case, although p can be found quickly, all possible periods are equivalent, and all possible partitions of the sequence for each candidate p are equivalent, thus there are $n^2 2^{n-1}$ total equivalent results. As a practical matter, we cut off the number of partitions to 20 per period, but with input strings that do have a component period, we do not hit the threshold in practice.

4 EXAMPLE TRIAL

To test the algorithm, we ran it on a small data set gathered by observing the behavior of a single individual retrieving coffee from the department coffee maker over the period of five days. This data set differs slightly from the type of data de- scribe above in that while it is repetitive, the time differences between each pass could be used to partition the passes correctly. However, if we ignore those breaks and run the data together into a single stream, then it does fit into the paradigm of the algorithm. The behavior sequence we observed correspond to two normal coffee retrieval events, and event where the pot was empty and needed to be remade, one normal coffee event, and a coffee retrieval at the end of the day, so the machine was shut off, for a total of 64 steps. Details of the data and the results can be found in (Crabbe, 2011).

In the first stage of the process, algorithm 1 correction identifies the normal coffee retrieval as the period. Because one of the variations from the period is long compared to the period itself, the algorithm 1 finds that the cost of considering the variation to be a single partition is equal to splitting the sequence in the middle and considering the first part as a suffix to one partition and the second as it's own. Algorithm 1 generates nine partitions, all of which differ in where

they split the "make new pot" event. Of those nine, all are converted by algorithm 2 into equal size FSTs, as measured by number of actions. A Lisp implementation of the algorithm, running on a single 3GHz Intel Core i7, generates the nine programs in 10.4 seconds.

5 DISCUSSION AND FUTURE WORK

Although the algorithm is theoretically sound, there are several areas of potential improvements and elements that need to be addressed to incorporate it in to practical robotic system.

5.1 Splitting Long Sequences

As seen in the coffee data, when a particular pass through the loop introduces a long chain of actions dissimilar to other passes, the partitioning algorithm generates multiple partitions of that pass, all having equal cost. This is especially common when there is an addendum at the end of a pass. The algorithm has no way to determine whether those actions are a suffix to the pass, a prefix to the next pass, or both (some actions are a suffix and some a prefix).

We believe that differentiation between the multiple resulting programs can best be done after their generation. Some FSTs may more naturally match the world state changes (see below). Another mechanism to differentiate would by to actually try the FSTs and measure their success. In the future we will examine both of these techniques.

5.2 Generalizing World State

As described above, the approach taken here is to attempt to infer a minimal program from only observed actions of another agent. We reapply the inputs in algorithm 2 to build the final FST. In that algorithm we make no assumption about how each set of inputs is related (e.g. the input for pass 0 may or may not intersect with that of pass 1). This approach misses some potential generalization in the resulting programs. For instance, if pass 0 and pass 1 have identical actions, it would make sense to attempt to generalize the world states at the start of both passes. Future work will include applying logical machine learn- ing techniques such as Version Space style learn- ing (Mitchell, 1982) on the input sets to generalize good descriptions of the states. This may have ramifications on the partitioning that takes place in our algorithm. As discussed above, if there are several equally good partitions, some may be eliminated based on the quality of the input generalization.

5.3 Data Assumptions

There are three primary assumptions made about the data required for the algorithm to work: the data is repetitive, that the input to the robot at the beginning of the loop is sufficient to determine the behavior through that pass, and that the correct period p occurs in the input string x. We maintain that these assumptions are reasonable for many kinds of robot tasks. Typical robot applications are for repetitive tasks, from manufacturing to maintenance to patrol. Many other tasks such as food preparation and service also consist of these sorts repetitive tasks, complete with input arriving at the start of the loop (e.g. the customer's order). Finally, with a large enough data set, the probability of the "common" sequence of actions for a task is high. Although there may exist a partition that results in fewer overall edits, we prefer that the period be selected from the data, as we believe that the best period would be one that was actually displayed.

For future work, we will investigate ways to loosen the assumptions on the data, allowing for nested loops or input to be checked at other fixed moments in execution. We will also attempt to characterize bounds on the size of the output compared to the true minimal machine.

6 CONCLUSIONS

This paper has presented an algorithm for inferring a program from repetitive data. It uses a two step approach, first partitioning the data by minimizing the edit distance between the proposed partitions, treating each partition as a separate pass through a loop, converting the passes to a FSM and then minimizing that machine. It has shown that under the assumptions that the data was generated from a program containing a single non-nested loop and that branches taken within the loop are determined by input at the start of the loop, we can find a nearly minimal program in $O(n^5)$, instead of exponential time for the general case.

ACKNOWLEDGEMENTS

The author would like to thank Chris Brown and Rebecca Hwa for many helpful insights. This work was

supported with a grant from the Office of Naval Research.

REFERENCES

Argall, B., Chernova, S., Veloso, M., and Brown-ing, B. (2009). A survey of robot learn- ing from demonstration. *Robotics and Au- tonomous Systems*, 57(5):469–483.

Bertsekas, D. P. (2005). *Dynamic Programming and Optimal Control*, volume 1. Athena Scientific, Belmont, MA.

Blumer, A., Ehrenfeucht, A., Haussler, D., and Warmuth, M. (1987). Occam's razor. *Information Processing Letters*, 24(6):377–380.

Bugalho, M. and Oliveira, A. (2005). Inference of regular languages using state merging algorithms. *Pattern Recognition*, 38:1457–1467.

Crabbe, F. (2011). Experiments on a technique for finding small fsts of repetitive unsegmented demonstration data. Usna-cs-2011-02, US Naval Academy.

Gold, E. M. (1978). Complexity of automoton identification from given data. *Iform. Control*, 37:302–320.

Hopcroft, J. (1971). *Theory of machines and computations*, chapter An n log n algorithm for minimizing states in a finite automaton, pages 189–196. Academic Press.

Lang, K. J., Pearlmutter, B. A., and Price, R. A. (1998). Results of the abbadingo one dfa learning competition and a new evidence-driven state merging algorithm. In *ICGI*, pages 1–12.

McNamara, J. J. (2010). 10 steps to load, stow and secure a freight container. *The Journal of Commerce*.

Mitchell, T. M. (1982). Generalization as search. *Artificial Intelligence Journal*, 18(2):203–226.

Pitt, L. and Warmuth, M. (1993). The minimum consistent dfa problem cannot be approximated within any polynomial. *J. Assoc. Comput. Mach.*, 40(1):95–142.

Sim, J. S., Iliopoulos, C. S., Park, K., and Smyth, W. (2001). Approximate periods of strings. *Theoretical Computer Science*, 262(557-568).

Veeraraghavan, H. and Veloso, M. M. (2008). Teaching sequential tasks with repetition through demonstration. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems(AAMAS)*, volume 3, pages 1357–1360.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS