Algorithms for Linguistic Robot Policy Inference from Demonstration of Assembly Tasks

Neil Dantam, Irfan Essa, and Mike Stilman

Abstract We describe several algorithms used for the inference of linguistic robot policies from human demonstration. First, tracking and match objects using the *Hungarian Algorithm*. Then, we convert Regular Expressions to Nondeterministic Finite Automata (NFA) using the *McNaughton-Yamada-Thompson Algorithm*. Next, we use *Subset Construction* to convert to a Deterministic Finite Automaton. Finally, we minimize finite automata using either *Hopcroft's Algorithm* or *Brzozowski's Algorithm*.

1 Introduction

The purpose of this technical report is the describe the algorithms used for the inference of robot control policies from human demonstrations. Hybrid system models are a powerful and effective approach for verifying and controlling robotic systems. However, the development of these models is usually a manual process requiring both mathematical and domain-specific expertise. Our pipeline for policy inference, Fig. 1, helps to automate the production of formal, verifiable models for robot control. To perform this inference, we combine and apply different existing algorithms from optimization and language theory. This report provides a consolidated description of the algorithms used to implement this inference pipeline.

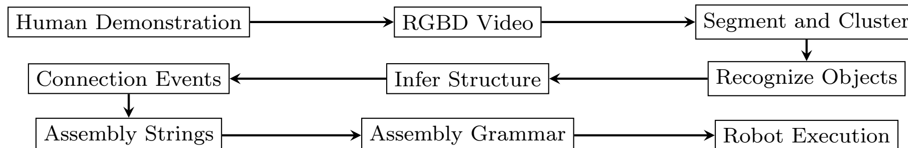

Fig. 1 Pipeline for automatic generation of grammars from human demonstration.

2 Object Tracking and Matching as The Assignment Problem

First, we track and match individual objects in the human demonstration video through the *Assignment Problem*. The Assignment Problem is an optimization problem that consists of finding the minimum cost matching between two sets, A and B, where the distances d between members of A and B are known..

Definition 1 (Assignment Problem). Given sets A and B and distance function $d : A \times B \mapsto \Re$, find the bijection $f : A \mapsto B$ such that $\sum_{a \in A} d(a, f(a))$ is minimized.

The Hungarian algorithm, shown in 1, is one approach for solving the assignment problem. This algorithm consists of the following seven steps.

Center for Robotics and Intelligent Machines, Georgia Institute of Technology, ntd@gatech.edu, irfan@cc.gatech.edu, mstilman@cc.gatech.edu.

Step 1 Subtract the minimum element from each row and column of the distance matrix.
Step 2 Star each zero in the distance matrix
Step 3 If every column contains a star, go to step 7. Otherwise go to step 4.
Step 4 Find an uncovered zero in the distance matrix. If there are no uncovered zeros, go to step 6. For any uncovered zero, prime the zero. If there is a star in the zero's row, uncover the star's column and look for the next uncovered zero. If there are no stars in the zero's row, go to step 5.
Step 5 Construct a through the distance matrix consisting of alternating stars and primes. First, find a starred column and add it to the path. If there are no more starred columns, the path is complete. Then find a primed row and add it to the path. If there are no more primed rows, the path is complete. When path is complete, unstar every starred element on the path, and star every unstarred element on the path. Go to step 3.
Step 6 Find the smallest uncovered value. Add this value to every covered row and subtract it from every uncovered column. Go to step 4.
Step 7 Each star now represents an optimum assignment.

The Hungarian algorithm is based on some assumptions about the structure of the assignment problem. First, we assume equal size sets A and B. If one of these sets is actually, smaller, we can pad the distance matrix with zeros to make the sets of equal size without affecting the optimum assignments. In addition, this algorithm computes the minimum cost assignment. To instead solve the maximum reward variation of the assignment problem, we can convert this to a minimum cost optimization by modifying the distance function as follows.

$$d'(a,b) = \max_{a' \in A, b' \in B} d(a',b') - d(a,b) \tag{1}$$

3 Regular Expressions to NFA

Later, we convert a Regular Expression to a Nondeterministic Finite Automaton (NFA) using the *McNaughton-Yamada-Thompson Algorithm* (MYT) [1, p159]. Regular Expressions and NFA are equivalent representations that both describe Regular languages.

The MYT algorithm, 2, performs a single top-down traversal of the parse tree for the Regular Expression. At each node of the parse tree, it adds the appropriate states and transitions. The algorithm in 2 is defined recursively. Thus, for the base case at the root of the parse tree, one initially starts with an empty NFA.

4 NFA to DFA

Next, we convert the NFA to an equivalent Deterministic Finite Automata (DFA). This conversion increases efficiency of the automaton in two ways. First, it is generally more efficient to execute a DFA on a computer than the to execute the NFA. This is because executing the DFA requires only tracking a single state at each point in time, while the nondeterminism of the NFA requires tracking multiple states. Second, we are able to compute minimum state equivalent DFAs, discussed in Sect. 5. This reduces both the cost of future operations on the automata and the storage requirements for the DFA's parsing table. We define this conversion algorithm based on [1, p152].

The NFA to DFA conversion uses two uses two subprocedures. The procedure ϵ-closure computes the set of all states reachable from some initial state via only ϵ transitions. The provided definition starts with some initial set of states S and some initial closure C. Then, for each ϵ transition from $s \in S$ to a state $s' \notin C$, it adds s' to C and recurses on s'. The procedure move-ϵ-closure computes the ϵ-closure of all states reachable from some initial set of states after a single non-ϵ transition.

The algorithm in 5 converts the NFA to a DFA. It operates by simulating the NFA and iteratively constructing subsets of the NFA states. For each possible transition, a new q', subset is constructed from the move-ϵ-closures of q'. The algorithm terminates when we have computed move-ϵ-closure for every created subset.

5 DFA Minimization

To reduce the computational requirements for policy execution, we minimize the state of the DFA using Brzozowski's Algorithm [2] or Hopcroft's Algorithm [4]. While Hopcroft's Algorithm provides superior worst case performance of $O(n \log(n))$ in number of states n compared to possibly exponential time for Brzozowski's, typical cases often fare better with Brzozowski [3].

Brzozowski's Algorithm, 6, operates by reversing all connections in the FA and converting the resulting NFA to a DFA, then repeating that procedure once more. The result is a DFA with minimum state.

Hopcroft's Algorithm, 7, operates by repeatedly partitioning the states of a DFA. Initially, it creates an initial partition Q' of states which are accepting and states which are rejecting. Then, it progressively refines Q' into subgroups which whose transitions for every symbol go only to states in the same subgroup of Q'. Finally, when no further refinements can be made, the algorithm terminates.

6 Summary

In this report, we described several algorithms used in the translation of human demonstrations to robot policies. The Hungarian algorithm for the Assignment Problem helps track and match objects in the demonstration. The McNaughton-Yamada-Thompson converts a Regular Expression for the policy to equivalent policy NFA. We use subset construction to convert the NFA to equivalent DFA. Finally, we minimize the state of the policy DFA using either Brzozowski's or Hopcroft's algorithm.

References

[1] A. Aho, M. Lam, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, & Tools.* Pearson, 2nd edition, 2007.

[2] J.A. Brzozowski. Canonical regular expressions and minimal state graphs for definite events. *Mathematical theory of Automata*, 12:529–561, 1962.

[3] J.M. Champarnaud, A. Khorsi, and T. Paranthoën. Split and join for minimizing: Brzozowskis algorithm. *Proc. of PSC*, 2: 96–104, 2002.

[4] J. Hopcroft. An n log n algorithm for minimizing states in a finite automaton. *Reproduction*, pages 189–196, 1971.

Algorithm 1: Hungarian Algorithm

```
   Input: D : matrix ℜ^n × ℜ^n ;                                   // Pairwise distances
   Output: A : list ℕ × ℕ ;                                        // Assignments
1  S ← FALSE ; // star mask, n × n boolean matrix
2  P ← FALSE ; // prime mask, n × n boolean matrix
3  r ← FALSE ; // row cover, n boolean vector
4  c ← FALSE ; // column cover, n boolean vector
   /* STEP 1: Subtract smallest element from rows and columns              */
5  forall j=1,n do
6      col(D, j) ← col(D, j) − min(col(D, j))
7  forall i=1,n do
8      row(D, i) ← row(D, i) − min(row(D, i))
   /* STEP 2: Star each zero in D                                          */
9  for i=1 to n do
10     if ∃j | 0 = D_{i,j} ∧ ¬r_i ∧ ¬c_j then
11         r_i ← TRUE ; c_i ← TRUE ; S_{i,j} ← TRUE ;
12 end ← FALSE ;
13 STEP3: repeat
       /* STEP 3: Check if all columns starred                             */
14     forall j=1,n do
15         c_j ← any(col(S, j)) ;
16     if count(c) = n then
17         end ← TRUE ;
18         break ;
       /* STEP 4                                                           */
19     P ← FALSE;
20     r ← FALSE;
21     STEP4: for i=1 to n do
22         if ∃j | 0 = D_{i,j} ∧ ¬r_i ∧ ¬c_j then
23             P_{i,j} ← TRUE;
24             if ∃k | S_{i,k} = TRUE then
25                 r_i ← TRUE ; c_k ← FALSE ; // Cover row, uncover col
26                 cycle STEP4 ;
27             else
                   /* Nothing starred in row                               */
28                 H_1 ← (i, j) ;
                   /* STEP 5                                               */
29                 b ← 1 for k = 1,n do
30                     if ∃i | S_{i,H_{b,2}} = TRUE then
31                         H_{b+1} = (i, H_{b,2}) ;
32                         b ← b + 1;
33                         if ∃j | P_{j,H_{b,1}} = TRUE then
34                             H_{b+1} = (H_{b,1}, j) ;
35                             b ← b + 1;
                   /* Convert path                                         */
36                 forall k = 1 to b do
37                     S_{H_k} = ¬S_{H_k} ;
38                 cycle STEP3 ;
           /* STEP 6                                                       */
           /* Find smallest uncovered value in D                           */
39         m ← min_{∀i|¬r_i, ∀j|¬c_j} D_{i,j} ;
40         forall i|r_j do
41             row(D, i) ← row(D, i) + m; /* Add to covered rows           */
42         forall j|¬c_j do
43             col(D, j) ← col(D, j) − m; /* Subtract from uncovered columns */
44 until end ;
   /* STEP 7: Compute Assignments                                          */
45 A ← {(i, j)|S_{i,j} = TRUE};
```

Algorithm 2: Recursive McNaughton-Yamada-Thompson Algorithm

Input: T, Q, E, s ; `// regex tree, NFA states, edges, start state`

Output: Q', E', a ; `// NFA states, edges, end state`

1. **if** *root(T) = CONCATENATION* **then**
2. $\quad (Q', E', a) \leftarrow$ **fold-left**$(\text{MYT}, (Q, E, s), \text{children}(T))$;
3. **else if** *root(T) = UNION* **then**
4. $\quad L \leftarrow$ **map**$(\lambda(T^*)\{MYT(T^*, Q, E, s)\}, \text{children}(T))$;
5. $\quad (Q', E', a) \leftarrow$ **fold-left**$(\lambda(Q^*, E^*, a^*)\{Q^*, E^* \cup (s \xrightarrow{\epsilon} a^*), a^*\}, \text{L})$;
6. **else if** *root(T) = KLEENE-CLOSURE* **then**
7. $\quad s_2 \leftarrow \text{newstate}()$;
8. $\quad (Q^*, E^*, a) \leftarrow \text{MYT}(\text{ children(T), Q, E, } s_2)$;
9. $\quad E' \leftarrow E^* \cup (s \xrightarrow{\epsilon} s_2) \cup (s \xrightarrow{\epsilon} a) \cup (a \xrightarrow{\epsilon} s_2)$;
10. $\quad Q' \leftarrow Q^* \cup s_2$;
11. **else**
12. $\quad a \leftarrow \text{newstate}()$;
13. $\quad E' \leftarrow E \cup (s \xrightarrow{T} a)$;
14. $\quad Q' \leftarrow Q \cup a$;

Procedure ϵ-`closure`(*NFA,S,C*)

Input: NFA, S, C ; `// NFA, initial states, initial closure`

Output: C' ; `// final closure`

1. $f \leftarrow \lambda(c, s)\left\{\textbf{if}(s \in c)\ c\ \textbf{else}\ \epsilon\text{−closure}\left(\text{NFA}, \bigcup_{s \xrightarrow{\epsilon} q \in \text{NFA}} q, s \cup c\right)\right\}$;
2. $C' \leftarrow$ **fold-left**(f, C, S) ;

Procedure `move-`ϵ`-closure`(*NFA,S,Z*)

Input: NFA, S, Z ; `// NFA, initial states, token`

Output: C' ; `// reachable states`

1. $f \leftarrow \lambda(c, s)\left\{\epsilon\text{−closure}\left(\text{NFA}, \bigcup_{s \xrightarrow{z} q \in \text{NFA}} q, c\right)\right\}$;
2. $C' \leftarrow$ **fold-left** $(f, \emptyset, \epsilon\text{−closure}(\text{NFA}, S, \emptyset))$;

Algorithm 5: NFA-to-DFA

Input: Q, Z, E, s, a ; `// NFA states, tokens, edges, start, accept`

Output: Q', Z', E', s', a' ; `// DFA states, tokens, edges, start, accept`

1. $Z' \leftarrow Z$;
2. $s' \leftarrow \epsilon - closure(s)$;
3. $Q' \leftarrow s'$;
4. $E' \leftarrow \emptyset$;
5. $i \leftarrow 0$;

`/* Construct Subsets */`

6. **while** $i < |Q'|$ **do**
7. $\quad$ **forall** $z \in Z$ **do**
8. $\quad\quad u = \text{move}{-}\epsilon{-}\text{closure}((Q, E), Q'_i, z)$;
9. $\quad\quad$ **if** u **then**
10. $\quad\quad\quad Q' \leftarrow Q' \cup u$;
11. $\quad\quad\quad E' \leftarrow E' \cup (Q'_i \xrightarrow{z} u)$;
12. $\quad i \leftarrow i + 1$;

`/* Accept States */`

13. $a' = \{q \in Q' | a \in q\}$

Algorithm 6: Brzozowski's Algorithm

```
  Input: Q, E, s, a ;                          // FA states, edges, start, accept
  Output: Q', E', s', a' ;                     // Minimum DFA states, edges, start, accept
1 nfa-to-dfa(reverse-fa(nfa-to-dfa(reverse-fa(Q,E,s,a))));
```

Algorithm 7: Hopcroft's Algorithm

```
   Input: Q, Z, E, s, a ;                      // FA states, tokens, edges, start, accept
   Output: Q', Z', E', s', a' ;                // Minimum DFA states, tokens, edges, start, accept
 1 Z' ← Z;
 2 Q' ← {a, Q − a} ; // Initial Partitioning
 3 T ← a ;
 4 while T do
 5     q' ← pop(T) ;
 6     forall z ∈ Z do
 7         x ← ⋃_{p -z-> q ∈ E, ∀q ∈ q'} p ; // All z predecessor states of partition q'
 8         if x then
 9             Q* = ∅;
10             forall y ∈ Q' do
11                 i = y ∩ x ; // Subset of partition y transitioning on z to q'
12                 j = y − x; // Subset of partition y transitioning on z to ¬q'
13                 if i ∧ j then
14                     Q* ← Q* ∪ i ∪ j; // Replace partition y with i and j
15                     if y ∈ T then
16                         T ← (T − y) ∪ j ∪ j;
17                     else if |i| < |j| then
18                         T ← T ∪ i;
19                     else
20                         T ← T ∪ j;
21                 else
22                     Q* ← Q* ∪ y; // Don't split y
23             Q' ← Q* ;
```