# Fixed Parameter Inductive Inference

Michael Wehar University at Buffalo mwehar@buffalo.edu

December 28, 2013

#### Abstract

We investigate a particular inductive inference problem where we are given as input a finite set of ordered triples of the form  $(w_i, b_i, t_i)$  such that  $w_i$  is a bit string,  $b_i$  is a single bit, and  $t_i$  is a positive integer written in unary. A machine Mis said to match such a triple if M outputs  $b_i$  on input  $w_i$  in at most  $t_i$  steps. We proceed to investigate the fixed parameter problems where we fix the number of triples. We show that the problem for k triples is in P. However, no deterministic machine is known to solve it in less than  $O(n^k)$  time. These problems are of particular interest in computational complexity theory because allowing  $O(\log(n))$ non-deterministic guesses offers a significant improvement from the best known deterministic algorithm.

## 1 Introduction

**Definition 1.1.** IIP will denote the following problem. Given as input a finite set of triples as described above and an integer s, does there exist a Turing machine M such that M matches each triple and M has at most s states.

The problem IIP is a member of a more general class of problems where we are given as input two disjoint sets of words and a list of auxiliary conditions. The goal is to find the smallest machine of a particular type that separates the words while meeting the auxiliary conditions. For IIP, the sets of words are finite sets, the type of machine

 $<sup>\</sup>ast$  The research work for this paper was done under the direction of Manuel Blum from Carnegie Mellon University.

is two tape Turing machine with a read only input tape and a read/write work tape, and the auxiliary conditions consist of runtime bounds for the words from the given two sets. It's important to notice that for Turing machines, if the input does not include auxiliary conditions, then the problem is not solvable. However, for DFA's and PDA's, the problem of separating two disjoint finite sets of words is solvable (without auxiliary conditions). A few such problems are discussed in [10]. In addition, there are problems that involve infinite sets of words. For example, finding the smallest DFA that separates two disjoint regular languages [2].

The problem IIP is clearly in NP. IIP is also believed to be NP-hard [1]. This seems to be a reasonable hypothesis given that finding the smallest DFA that separates two finite sets of words is known to be NP-hard [5]. However, after much effort, the author decided to take the opposite route and try to find methods for solving the problem. In particular, we present a construction that assigns each finite set of triples a sufficiently small Turing machine. The number of states in the assigned Turing machine will depend more so on the number of triples than the length of the  $w_i$ 's. We are therefore led to consider the fixed parameter problems where we fix the number of triples. Let k-IIP denote the set of strings in IIP with at most k triples. First, we give a construction for matching two triples. Then, we optimize the construction to show that 2-IIP  $\in$  P. Finally, we generalize the construction to show that k-IIP  $\in$  P for all  $k \in \mathbb{N}$ .

### 2 Basic Construction

**Proposition 2.1.** 2-IIP  $\in$  DTIME(poly(n)  $\cdot 2^{O(\log(n) \log \log(n))})$ .

Proof. Let an input consisting of  $(w_1, b_1, t_1)$ ,  $(w_2, b_2, t_2)$ , and s be given. First, if  $b_1 = b_2$ , then there exists a machine with one state that matches both triples i.e. either the machine that accepts all words or the machine that rejects all words. The remainder of the proof will be concerned with the case  $b_1 \neq b_2$ .

Consider the words  $w_1$  and  $w_2$ . Let *i* denote the least bit index where  $w_1$  and  $w_2$  differ. In the event that  $w_1$  is a prefix of  $w_2$  or  $w_2$  is a prefix of  $w_1$ , *i* will denote the length of the shorter word. It is important to note that the Turing machines have blank symbols after the input word so that it can recognize when the end of the input word has been reached. Also, consider the runtimes  $t_1$  and  $t_2$ . If  $t_1$  or  $t_2$  is less than *i*, then there does not exist a machine that matches both triples because it takes at least *i* steps for the input head to get to a bit where  $w_1$  and  $w_2$  differ. Therefore, it is appropriate for us to say that a machine matches both triples with optimal runtime if it halts on



2.a. Three Stages.

2.b. Diagram for printing 1101.

inputs  $w_1$  and  $w_2$  in exactly *i* steps. Such a machine will also be said to have optimal runtime on  $w_1$  and  $w_2$ .

We will construct a machine that matches both triples with optimal runtime and this machine will have  $\lceil 3\log(i) \rceil + c$  states where c is a constant that does not depend on the triples. Therefore, if s is at least  $\lceil 3\log(i) \rceil + c$ , then there exists a machine with at most s states that matches both triples. Otherwise, since for all  $m \in \mathbb{N}$  there are  $2^{O(m\log(m))}$  machines with m states, we get an upper bound  $2^{O(\log(i)\log\log(i))}$  for the number of machines with at most s states. Therefore, we can solve 2-IIP with brute force enumeration in DTIME(poly(n)  $\cdot 2^{O(\log(n)\log\log(n))}$ ) steps.

**Claim 2.1.** There exists a Turing machine with at most  $\lceil 3 \log(i) \rceil + c$  states that matches both triples and has optimal runtime on  $w_1$  and  $w_2$ .

Proof. We want our machine to move across the input tape to index i where  $w_1$  and  $w_2$  differ and decide which bit to output based on which bit is stored at index i. Let's start with an easier construction for a machine with  $\lceil \log(i) \rceil + c$  states that doesn't meet the optimal runtime condition. We build a machine that prints the number i in binary and then decrements the value on the work tape like a counter until it reaches 0. Each time the machine decrements the value on the work tape, the input head shifts one bit to the right. When the work tape value reaches 0, the input head will be at index i and the machine will decide which bit to output. The machine can be made to run in  $(2i + 1) \lceil \log(i) \rceil$ . It takes  $\lceil \log(i) \rceil$  steps to print i and  $2i \lceil \log(i) \rceil$  steps to decrement the counter from i to 0.

Now, we will show how to modify the preceding construction to meet the optimal runtime condition. If *i* has the form  $\lceil \log(j) \rceil (2j+1)$  for some  $j \in \mathbb{N}$ , then we can build a machine that prints *j* and then decrements such that at each step the input head moves 1 index to the right. Since this machine runs in *i* steps, the work tape value will be 0 when the input head is at index *i*. The number of states in this machine is  $\lceil \log(j) \rceil + c$ . For an arbitrary  $i \in \mathbb{N}$ , we can find  $j \in \mathbb{N}$  such that  $(2j+1)\log(j) \leq i \leq (2j+3)\log(j+1)$ .

#### Michael Wehar

Using calculus, we get  $(2j+3)\log(j+1) - (2j+1)\log(j) = \int_{j}^{j+1} (2\log(x) + 2 + \frac{1}{x}) dx \le 2\log(j+1) + 3$ . Therefore, *i* is within  $2\log(j+1) + 3$  of  $(2j+1)\log(j)$ . Hence, we can take the machine for  $(2j+1)\log(j)$  with  $\lceil \log(j) \rceil + c$  states and add at most an additional  $2\log(j+1) + 3$  states to get a machine for *i* with at most  $\lceil 3\log(j) \rceil + c \le \lceil 3\log(i) \rceil + c$  states.  $\Box$ 

# **3** Optimization for Smaller Machine

#### Theorem 3.1. 2-IIP $\in$ P.

Proof. In the previous argument, we constructed a small machine that matches two triples. Given two triples with input words  $w_1$  and  $w_2$ , the machine moves to the first bit index *i* where  $w_1$  and  $w_2$  differ and outputs based on the bit at this index. The machine's computation is split into three stages represented by figure 2.a. The machine has approximately  $\log(i)$  states. I claim that there is a smaller machine that moves to index *i* and halts. We will construct such a machine with h(i) states such that  $2^{h(i)\log(h(i))}$  is bounded by a polynomial in *i*. This will imply that we only need to enumerate through polynomially many machines to find the smallest machine that matches two triples. Therefore, 2-IIP  $\in$  P.

The function h is defined by  $h(i) = max\{ j \in \mathbb{N} \mid 2^{j \log(j)} \leq i \}$ . We will introduce a construction for a machine with O(h(i)) states that moves to index i and then halts. Similar to before, the machine will print a bit string on the work tape, decrement its value, and finally move the input head a remaining number of cells to the right.

Let a positive integer m be given. We will give a construction for a machine with O(h(m)) states that prints m on the work tape in at most p(h(m)) steps for some polynomial p. Let  $S_j$  for j from 1 to h(m) denote states. In addition, O(h(m)) auxiliary states will be used to perform predetermined subroutines. There will be type 1 transitions from  $S_j$  to  $S_{j+1}$  and  $S_{j+1}$  to  $S_j$  for each j. Each state  $S_j$  will be the source of one type 2 transition. There are h(m) choices for each type 2 transition's target. Therefore, we can represent each choice by a bit string of length  $\log(h(m))$ . We can represent a full list of choices, one per state, by a bit string of length  $\log(m) = h(m) \log(h(m))$ . In addition, each bit string is associated with such a list of choices. Therefore, there is a list of choices that is associated with the binary representation of m.

Next, we will describe how the machine prints m. The process consists of h(m) phases, one for each state. In phase j, the machine will print a bit string of length



3.a. Diagram for printing 10111111.

 $\log(h(m))$  that represents which choice  $S_j$  made (the target of its type 2 transition) and then move on to the next phase. In order to do this, the machine will need to keep count of which phase it is in and the bit strings printed in the previous phases. The machine will use the count for the current phase to move to the correct state via type 1 transitions. Then, it will follow a type 2 transition. Finally, it will follow type 1 transitions all the way back to  $S_1$ . As it moves back to  $S_1$  it will print the bit string for the associated choice. When it's finished, it will increment the phase count by 1. After all phases are complete, the work tape will have printed a bit string of length  $\log(m) = h(m)\log(h(m))$ . Now, it will decrement it's value. If at each step of the computation it moves one index to the right on the input tape, then when the counter reaches 0, it will be at an index between mh(m) and mh(m)+p(h(m)) for some polynomial p.

Now, we pick the largest m such that the machine from the above construction halts on an index less than i. There exists some polynomial q such that  $m \leq i \leq$ m+q(h(m)) for all i and m defined in the preceding manner. By the construction from proposition 2.1, there is a machine that moves the input head i - j indexes with at most  $\log(q(h(m)))$  states. If we combine these two machines, then we get a machine with O(h(i)) states that moves to index i and then halts.

### 4 Generalization for More Triples

#### Theorem 4.1. k-IIP $\in$ P.

*Proof.* Let k triples be given. Group input words from these triples into sets A and R such that A contains words to be accepted and R contains words to be rejected. In the following, we will construct a graph that represents the fewest number of characters

a Turing machine needs to read to determine whether a word from  $A \cup R$  is in A or R.

We will construct a directed labeled graph similar to a DFA that will represent which words are in A and which words are in R. The underlying directed graph structure is a rooted binary tree directed towards the leaves. The directed edges are labeled with 0 or 1. Vertices are labeled as accept or reject to represent that the directed path from the root to this vertex is associated with a word in A or R, respectively. This structure would be a DFA, but it's incomplete in the sense that some vertices are not labeled as accept or reject because some words are neither in A nor R. Also, there are some vertices that do not have an outgoing edge for 0 or 1. However, the structure at least determines for each word  $w \in A \cup R$  whether w is in A or R.

Up to this point, there could be several graphs that represent A and R in this manner. However, we want a particular minimal such graph that reads the fewest characters needed to determine whether a word is in A or R. The following condition allows us to achieve this. If a vertex is labeled with accept [reject] and no descendants are labeled with reject [accept], then the vertex has a loop back to itself and no children. However, the loop isn't really used because as soon as we get to a vertex with a loop, we already know whether to accept or reject the word and don't need to read any more characters. I claim that there is a single graph that satisfies the preceding description with the fewest number of vertices, directed edges, and labels.

Now, we will compress the preceding graph and use the resulting structure to build a Turing machine that picks out exactly which bits matter in determining whether an input word is in A or R. We will take the preceding graph, remove branches, and add labels denoting the size of the removed branch. In particular, each maximal branch that only contains unlabeled vertices with exactly one child will be removed and the directed edge leading to this branch will additionally be labeled with the number of the removed vertices. To avoid confusion, the additional labels will be enclosed by parentheses. The only exception is when the root has only one child. In this case, we can compress the maximal branch down to one directed edge.

Let's try to get an upper bound on the number of vertices in the graph. Since  $A \cup R$  contains at most k vertices, at most k vertices are labeled as accept or reject. It follows that there are at most k vertices with fewer than two children each. One can show by induction that every binary tree with at most k vertices with fewer than two children has at most 2k - 1 vertices. In addition, there are at most 2k - 2 edges since each vertex, besides the root, has an indegree of 1.

The Turing machine will have the structure of the graph that we constructed, but



4.a. Example of graph construction.

with the counter mechanisms described in theorem 3.1 implemented at each edge with a parentheses label to move the specified number of cells to the right using at most h(n) states. The machine will have at most (2k-2)h(n) + 1 states because the graph has one root and at most 2k - 2 edges for which we need to add at most h(n) states. In addition, the graph was constructed in such a way that the machine will separate Aand R with optimal runtime. Therefore, k-IIP is solvable in  $O(\operatorname{poly}(n) \cdot 2^{ckh(n)\log(kh(n))})$ steps for some constant c. Since k is fixed, k-IIP  $\in \mathbb{P}$ .

We will describe how the Turing machine operates as follows. The machine will take an input and follow the graph we constructed. We start at the root of the graph. If we are at a vertex with two children, then we decide which child to move to based on the current bit on the input tape and move the input head one cell to the right. If the edge we take also has a parentheses label, then we additionally move the input head the specified number of cells to the right. If we are at a vertex with only one child, then we check if we are at the end of the input head. If we are, then we accept or reject based on the label of the current vertex. Otherwise, we move the input head one cell to the right or one plus the specified number of cells if the edge has a parentheses label. If we are at a leaf, then we accept or reject based on the label of the current vertex.  $\Box$ 

#### 5 Conclusion

We investigated an inductive inference problem that if solvable quickly, can be used to compute the size of the smallest Turing machine that matches a given set of data. The author has particular interests in two applications:

(1) "Given a large bit string, can we give a short description for the string?"

#### Michael Wehar

(2) "Given a physical device that takes input and produces output, is there a small Turing machine that models this device?"

Application (1) comes up when compressing data or trying to infer sequences. Application (2) comes up when trying to computationally model the outcomes from a physical experiment [1]. The author is interested in how modeling human experiments by Turing machines can be useful. Consider a student who is taking a multiple choice test. Associate with each question a triple of the form (question, response, response time). Now, construct a Turing machine that matches all such triples that we get from monitoring the student. It may be difficult to determine how much temporary memory the student used to answer a particular question. However, it's easy to compute how much temporary memory a Turing machine uses on an input.

On the other hand, if IIP is not solvable more quickly, then  $P \neq NP$ . If the *k*-IIP problems are not solvable more quickly, then we can show that even a small amount of non-determinism can provide quicker algorithms. In particular, using the construction from theorem 4.1, we can show  $\exists c \in \mathbb{N} \ \forall k \in \mathbb{N}$ , *k*-IIP can be solved in  $O(n^c)$  steps using at most  $O(\log(n))$  non-deterministic guesses. However, if  $\nexists c \in \mathbb{N}$  $\forall k \in \mathbb{N}$ , *k*-IIP  $\in$  DTIME $(n^c)$ , then a small amount of non-determinism provides a significant improvement for the *k*-IIP problems. More formally, we would get that for every non-decreasing function  $g(n) = \omega(\log(n))$  that can be computed in polynomial time, there is a problem solvable by a non-deterministic machine in polynomial time using at most g(n) non-deterministic guesses that is not in P. The reader should notice that IIP  $\notin$  FPT implies  $\nexists c \in \mathbb{N} \ \forall k \in \mathbb{N}$ , *k*-IIP  $\in$  DTIME $(n^c)$ .

It is also important to consider how difficult IIP is in comparison to the DFA problem. In particular, let  $\mathcal{D}$  denote the problem for DFA's where we are given two disjoint finite sets of words and a number s and want to determine if there exists a DFA of size at most s that separates the two sets. It was shown in [5] that  $\mathcal{D}$  is NP-hard. We were unable to show that IIP is NP-hard. However, we did show that k-IIP  $\in$  P. Consider k- $\mathcal{D}$  where we restrict our attention to inputs where the total number of words is k. We are led to the question, "Is 2- $\mathcal{D}$  in P or NP-hard?" One might expect that 2- $\mathcal{D}$  is in P because 2-IIP is in P and Turing machines are more computationally powerful than DFA's. However, the fact that Turing machines are more computationally powerful than DFA's should make the problem easier because we may be able to use their computational power to build smaller machines that separate finite sets. The question is open and we refer the interested reader to [3] and [10].

As a final remark, we were unable to show that the k-IIP are solvable in less than linear space. However, we suggest that one may be able to apply results from [7] to show that k-IIP  $\in$  DSPACE $(\frac{n}{\log(n)})$ . Since k-IIP  $\in$  P, either there exist algorithms for solving k-IIP in less space or P  $\neq$  L.

### 6 Acknowledgments

The author would like to thank Manuel Blum for several meetings and discussions focused on IIP and other inductive inference problems. In addition, the author would like to thank Manuel Blum and Joseph Swernofsky for suggestions and corrections related to the content of the paper.

# References

- [1] Manuel Blum. *Timed Inductive Inference*. Unpublished, 2012.
- [2] Yu-Fang Chen, Azadeh Farzan, Edmund M. Clarke, Yih-Kuen Tsay, and Bow-Yaw Wang. Learning minimal separating dfas for compositional verification. *Lecture Notes in Computer Science*, 5505:31–45, 2009.
- [3] James Currie, Holger Petersen, John Michael Robson, and Jeffrey Shallit. Separating words with small grammars. J. Automata, Languages, and Combinatorics, 4:101–110, 1999.
- [4] Erik D. Demaine, Sarah Eisenstat, Jeffrey Shallit, and David A. Wilson. Remarks on separating words. CoRR, abs/1103.4513, 2011.
- [5] E Mark Gold. Complexity of automaton identification from given data. Information and Control, 37(3):302 – 320, 1978.
- [6] Gregor Gramlich and Ralf Herrmann. Learning unary automata. Presented at Descriptional Complexity of Formal Systems, 2005.
- [7] John Hopcroft, Wolfgang Paul, and Leslie Valiant. On time versus space. J. ACM, 24(2):332–337, April 1977.
- [8] Leonard Pitt and Manfred K. Warmuth. The minimum consistent dfa problem cannot be approximated within any polynomial. J. ACM, 40(1):95–142, January 1993.

- [9] J. M. Robson. Separating words with machines and groups. RAIRO Theoretical Informatics and Applications - Informatique Thorique et Applications, 30(1):81– 86, 1996.
- [10] Jeffrey Shallit. The separating words problem. *Presented at McMaster University* optimization seminar, 2010.