

# The Nature of Search Algorithms

ALBERT Y.S. LAM & VICTOR O.K. LI  
Department of Electrical & Electronic Engineering  
The University of Hong Kong

Technical Report TR 2009-01

(1<sup>st</sup> version Jan 29, 2009; revised Jun 26, 2009)

---

Problem-solving is one of the most important intellectual activities of human beings and most of the problems require us to find a desirable solution among a large set of solutions. Search algorithms are developed to solve these problems. However, a search algorithm which is excellent at solving a particular problem may perform poorly at another problem. Until the discovery explained in this article, we do not know why a particular search algorithm works well for a particular problem; we keep on developing algorithms according to our experiences, intuition, and by trial-and-error. This work is dedicated to exploring the nature of search algorithms. We discover that all search algorithms are composed of some basic units and all these units are finite and can be listed explicitly. Our discovery allows us to characterize the desirable properties of an algorithm to solve a particular problem. We believe this discovery allows us to have a better understanding of search and optimization theory.

General Terms: Algorithms, Performance, Theory

Additional Key Words and Phrases: Search, optimization, problem solving, enumeration

---

## 1. INTRODUCTION

Problem-solving is one of the most important intellectual activities of human beings. Solving the problems encountered in our daily lives and in research typically involves the technique of searching, i.e. finding a solution to satisfy all the requirements specified in the problem from among many others. For example, in optimization problems, we desire to obtain the best solution, whose objective function value is either maximized or minimized, among all possible feasible solutions. Such optimization problems exist in fields ranging from profit maximization in economics to signal interference minimization in electrical engineering. We classify all these searching techniques as search algorithms, and they cover most of the algorithms studied in computer science.

We can tackle simple search problems by intuition, while the more complicated ones can usually be solved only with mathematics and logics. Unfortunately, most problems are of the latter type. Thus, we always have to formulate problems in terms of equations

This research was supported by the University of Hong Kong Strategic Research Theme on Information Technology.

Authors' addresses: A. Y.S. Lam and V. O.K. Li, Department of Electrical and Electronic Engineering, Chow Yei Ching Building, The University of Hong Kong, Pokfulam Road, Pokfulam, Hong Kong.

or formulae in order to bring the powerful mathematics and logics into full play. In many cases, we rely on computers to execute these algorithms.

No single search algorithm is universal; there is no algorithm which can be applied to solve all kinds of search problems. This explains why computer scientists and engineers have been spending great efforts in developing search algorithms since the invention of the computer. An algorithm can give good performance only when matched with the right type of problem.

In this context, there are three kinds of objects, i.e. algorithms, computers, and problems, interacting with one other. Understanding their relationships can facilitate the matching, and thus, more problems can be addressed efficiently. Generally, we know about computers and problems much better than algorithms. In this article, we show our discovery on the nature of search algorithms.

## 2. BACKGROUND

### 2.1 Computers

Generally, a computer is a machine, with storage to save data, which manipulates specific data according to the instructions provided. There are two properties for all of today's computers: digital notation and finite cardinality. Everything stored in or executed by a computer is digital; everything is represented by a sequence of "0"s and "1"s. Moreover, the storage capacity is limited. This implies that the total number of representable objects is finite. We will never have a computer with infinite storage. To summarize, everything representable in a computer is discrete and finite.

### 2.2 Problems

We always describe problems as functions because this allows us to use mathematics and logics. We replace "problems" with "functions" throughout this article. Let  $f$ ,  $x$  and  $y$  denote a function, the input and the output of the function, respectively. Their corresponding sets are represented by  $F$ ,  $X$  and  $Y$ . Thus,  $f(x) = y$  for any  $x \in X$ ,  $y \in Y$  and  $f \in F$ , where  $x$  is a vector whose length can be any finite natural number and defines the dimension of  $f$ , while  $y$  is a scalar.  $|X|$ ,  $|Y|$  and  $|F| = |Y|^{|X|}$  stand for the cardinalities or sizes of the three sets, respectively.  $x$  represents a solution to  $f$  while  $y$  gives the performance evaluated according to  $f$ . The objective is to find the  $x \in X$  with a desirable  $y$  value.

We can consider a function as a “black-box” generating a unique output for each input. We always describe a function in the literature with a formula, e.g.  $y = x+2$ , and we can also specify a function as a list of  $(x,y)$  pairs. However, when only given the formula of an unfamiliar function, we can only determine the corresponding  $y$  for each  $x$  once at a time. We cannot tell everything about the function solely with the formula. We can do so only if we have all  $(x,y)$  pairs of the function. Therefore, we re-define “knowing” of functions with this interpretation.

Computers are the platform on which algorithms compute the functions. Thus, functions characterized in computers must be discrete and finite. Moreover, all elements belonging to a set are well-ordered because a computer describes everything in terms of a sequence of “0”s and “1”s. To specify a set of objects, we need to pre-define how much memory, in terms of number of bits, will be allocated for its elements. As the elements are given as sequences of “0”s and “1”s of the same number of bits, they may be compared with each other and ordered as a one-dimensional list. However, elements from different sets may not be compared because they are denoted by sequences of different number of bits. Without loss of generality,  $X$ ,  $Y$  and  $F$  can be described by their index sets and stated as  $\{1,2,\dots,|X|\}$ ,  $\{1,2,\dots,|Y|\}$  and  $\{f_1, f_2, \dots, f_{|Y|^{|X|}}\}$ , respectively. For example, if

TABLE I. The list of all possible functions for  $X=\{1,2,3\}$  and  $Y=\{1,2,3\}$ .

Table entries are $y$ - components of the histories		$x$ -values		
		1	2	3
Functions	$f_1$	1	1	1
	$f_2$	2	1	1
	$f_3$	3	1	1
	$f_4$	1	2	1
	$f_5$	2	2	1
	$f_6$	3	2	1
	$f_7$	1	3	1
	$f_8$	2	3	1
	$f_9$	3	3	1
	$f_{10}$	1	1	2
	$f_{11}$	2	1	2
	$f_{12}$	3	1	2
	$f_{13}$	1	2	2
	$f_{14}$	2	2	2
	$f_{15}$	3	2	2
	$f_{16}$	1	3	2
	$f_{17}$	2	3	2
	$f_{18}$	3	3	2
	$f_{19}$	1	1	3
	$f_{20}$	2	1	3
	$f_{21}$	3	1	3
	$f_{22}$	1	2	3
	$f_{23}$	2	2	3
	$f_{24}$	3	2	3
	$f_{25}$	1	3	3
	$f_{26}$	2	3	3
	$f_{27}$	3	3	3

$X=\{1,2,3\}$  and  $Y=\{1,2,3\}$ , then  $|F|=3^3=27$ . We can enumerate all  $f \in F$  as in Table I. In the table, each function is defined by the  $y$ -values corresponding to the  $x$ -values. For example, the row of  $f_2$  says that  $f_2(1)=2$ ,  $f_2(2) = f_2(3) = 1$ . Although  $x$  can be multi-dimensional, it is still represented as a sequence of “0”s and “1”s. Thus, we can make any  $x$  one-dimensional. For example,  $x = (x_1, x_2) = (3,5)$ . If we allocate three bits for both  $x_1$  and  $x_2$ , it can be described as a two-dimensional tuple (011,101), which can be further transformed into a one-dimensional 011101 by concatenating the two components of the two-dimensional tuple. Moreover, all possible constrained optimization problems can also be included in  $F$ . This can be done by specifying one  $y$  in  $Y$  as “undetermined”.

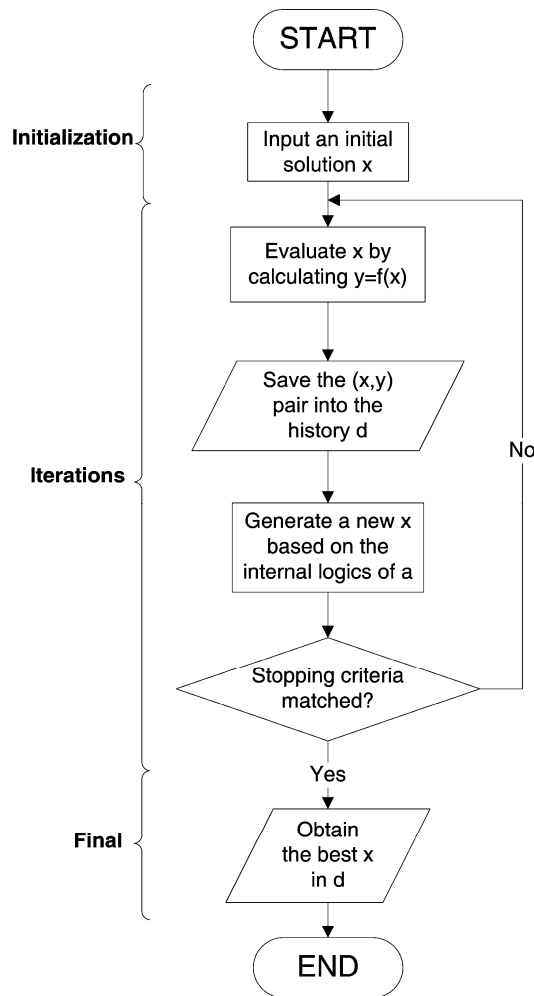


FIG. 1. Flow chart of the model of algorithm. START and END indicate the beginning and termination of each run of the model. In each run, we start with the initialization, perform a certain number of iterations, and terminate at the final stage.

### 2.3 Algorithms

An algorithm provides a way to search the  $x \in X$  with a desirable  $y$  value according to  $f$ . According to [Cormen et al. 2001], “an algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values as output.” So we can consider a search algorithm as a sequence of instructions which converts the input to the output. An algorithm is well-defined when it can generate an output for any possible input. Here we focus on an individual-based deterministic algorithm (see Figure 1) which performs iteratively. “Individual-based” means the algorithm generates a single output in each iteration as the input for the next

TABLE II. A summary of the variables manipulated in the model of algorithm.

Iteration	1	2	...	$m$	...	$ \mathcal{X}  - 1$
Input to the iteration	Non-algorithm-generated $x_1$	$x_2$	...	$x_m$	...	$X_{ \mathcal{X} -1}$
Function value	$f(x_1) = y_1$	$f(x_2) = y_2$	...	$f(x_m) = y_m$	...	$f(x_{ \mathcal{X} -1}) = y_{ \mathcal{X} -1}$
History	$d_1 = [(x_1, y_1)]$	$d_2 = [(x_1, y_1), (x_2, y_2)]$	...	$d_m = [(x_1, y_1), (x_2, y_2), \dots, (x_m, y_m)]$	...	$d_{ \mathcal{X} -1} = [(x_1, y_1), (x_2, y_2), \dots, (x_{ \mathcal{X} -1}, y_{ \mathcal{X} -1})]$
Output from the iteration	$a(d_1) = x_2$	$a(d_2) = x_3$	...	$a(d_m) = x_{m+1}$	...	$a(d_{ \mathcal{X} -1}) = x_{ \mathcal{X} }$

iteration, whereas “deterministic” means that the algorithm does not involve randomness. In other words, it generates an identical output whenever given the same input. Generally, in iteration  $m$ ,  $x_m$ , which is the output of the previous iteration, is put into the function  $f$ , generating output  $y_m$ . This  $(x_m, y_m)$  pair is then appended to all the previously generated pairs as a history  $d_m = [(x_1, y_1), (x_2, y_2), \dots, (x_m, y_m)]$ , i.e.,  $(x_m, y_m)$  is concatenated to  $d_{m-1}$  to form  $d_m$ . We identify the internal logics<sup>1</sup> of an algorithm as  $a$ . Such internal logics distinguish one algorithm from another.  $a$  manipulates  $d_m$  and outputs  $x_{m+1}$ , as the input to the next iteration. We assume  $x_{m+1}$  is different from all  $x$  already in  $d_m$ . We adopt the oracle-based view of computation [Wolpert and Macready 1995; Wolpert and Macready 1997] in the analysis. We compare the performance of an algorithm by checking the total number of function evaluations (calls) used to get the desired result. Moreover, we do not consider “revisited”  $x$  and try to distinguish an algorithm by looking at distinct “visited”  $x$ . For any algorithm  $a$  which may output previously generated  $x$  in some iterations, we can produce a corresponding algorithm  $a'$  whose generated histories are equal to those of  $a$  with all duplicated  $(x, y)$  pairs skipped. One of the most important theorems in search and optimization theory, the No-Free-Lunch theorem [Wolpert and Macready 1995; Wolpert and Macready 1997] was also derived from this model. This is possible if the system can memorize all the previously generated  $(x, y)$  pairs. We assume that  $a$  is applied to solve  $f$  in an ideal machine which has limited but sufficient storage to store  $X$  and  $Y$ , and all

<sup>1</sup> Internal logics refer to mechanisms which make an algorithm functional. We have not yet identified what internal logics are. But they will become clearer later in the article.

TABLE III. Enumeration of all possible histories generated by all algorithms for  $X=\{1,2,3\}$  and  $Y=\{1,2,3\}$ .

Table entries are y-components of the histories		x-components of the histories					
		3 2 1	3 1 2	2 3 1	2 1 3	1 2 3	1 3 2
Functions	$f_1$	1 1 1	1 1 1	1 1 1	1 1 1	1 1 1	1 1 1
	$f_2$	1 1 2	1 2 1	1 1 2	1 2 1	2 1 1	2 1 1
	$f_3$	1 1 3	1 3 1	1 1 3	1 3 1	3 1 1	3 1 1
	$f_4$	1 2 1	1 1 2	2 1 1	2 1 1	1 2 1	1 1 2
	$f_5$	1 2 2	1 2 2	2 1 2	2 2 1	2 2 1	2 1 2
	$f_6$	1 2 3	1 3 2	2 1 3	2 3 1	3 2 1	3 1 2
	$f_7$	1 3 1	1 1 3	3 1 1	3 1 1	1 3 1	1 1 3
	$f_8$	1 3 2	1 2 3	3 1 2	3 2 1	2 3 1	2 1 3
	$f_9$	1 3 3	1 3 3	3 1 3	3 3 1	3 3 1	3 1 3
	$f_{10}$	2 1 1	2 1 1	1 2 1	1 1 2	1 1 2	1 2 1
	$f_{11}$	2 1 2	2 2 1	1 2 2	1 2 2	2 1 2	2 2 1
	$f_{12}$	2 1 3	2 3 1	1 2 3	1 3 2	3 1 2	3 2 1
	$f_{13}$	2 2 1	2 1 2	2 2 1	2 1 2	1 2 2	1 2 2
	$f_{14}$	2 2 2	2 2 2	2 2 2	2 2 2	2 2 2	2 2 2
	$f_{15}$	2 2 3	2 3 2	2 2 3	2 3 2	3 2 2	3 2 2
	$f_{16}$	2 3 1	2 1 3	3 2 1	3 1 2	1 3 2	1 2 3
	$f_{17}$	2 3 2	2 2 3	3 2 2	3 2 2	2 3 2	2 2 3
	$f_{18}$	2 3 3	2 3 3	3 2 3	3 3 2	3 3 2	3 2 3
	$f_{19}$	3 1 1	3 1 1	1 3 1	1 1 3	1 1 3	1 3 1
	$f_{20}$	3 1 2	3 2 1	1 3 2	1 2 3	2 1 3	2 3 1
	$f_{21}$	3 1 3	3 3 1	1 3 3	1 3 3	3 1 3	3 3 1
	$f_{22}$	3 2 1	3 1 2	2 3 1	2 1 3	1 2 3	1 3 2
	$f_{23}$	3 2 2	3 2 2	2 3 2	2 2 3	2 2 3	2 3 2
	$f_{24}$	3 2 3	3 3 2	2 3 3	2 3 3	3 2 3	3 3 2
	$f_{25}$	3 3 1	3 1 3	3 3 1	3 1 3	1 3 3	1 3 3
	$f_{26}$	3 3 2	3 2 3	3 3 2	3 2 3	2 3 3	2 3 3
	$f_{27}$	3 3 3	3 3 3	3 3 3	3 3 3	3 3 3	3 3 3

possible histories<sup>2</sup>.  $a$  terminates either when all  $x$  in  $X$  have been exhausted (i.e. we have finished the  $(|X|-1)^{\text{th}}$  iteration), or when any of the stopping criteria is satisfied, e.g., the maximum amount of CPU time used, the maximum number of iterations performed, etc. We summarize all the variables used in the process in Table II. In fact, the above process, which is similar to the HIRE-ASSISTANT algorithm in [Cormen et al. 2001], serves as a model for developing algorithms. For all practical purposes, all algorithms, ranging from sorting (e.g. bubble sort, [Cormen et al. 2001]) to optimization algorithms (e.g. simulated annealing [Kirkpatrick et al. 1983]), can fit into this model. Let  $D_m$  be the set of all possible histories of length  $m$ , and  $D_m^x$  and  $D_m^y$  be the sets of all possible  $x$ -components

<sup>2</sup> We will show the set of all possible histories is finite in the next section.

and  $y$ -components of  $D_m$ , respectively, and  $m = 1, 2, \dots, |X|$ . Then  $D_{|X|}$  is the set of all possible complete histories<sup>3</sup> and  $|D_{|X|}^x|$  equals  $|X|!$ . For  $X = \{1, 2, 3\}$  and  $Y = \{1, 2, 3\}$ , as in Table I, we list all possible  $x$ -components of complete histories,  $d_{|X|}^x \in D_{|X|}^x$ , in the second row of Table III. We also show the enumeration of  $D_{|X|}$  (i.e. all possible histories computed on all defined functions) in Table III. The functions are defined, as in Table I, in terms of  $(x, y)$  pairs in ascending order of values of  $x$ , i.e.,  $[1 \ 2 \ 3]$ . The  $x$ -components of the histories give all possible permutations of  $X$ . There are six such permutations and they form the headings of six columns in the table. We list all possible histories  $d_3$  for each permutation. There are  $|X|! |Y|^{|X|} = 3! 3^3 = 162$  possible histories in this example. For example, the row for  $f_2$  shows that for  $d_3^x = [3 \ 2 \ 1]$ , i.e., the first permutation,  $d_3^y = [1 \ 1 \ 2]$ , or  $d_3 = [(3, 1), (2, 1), (1, 2)]$ .

From this perspective, we can observe that the “behavior” of an algorithm is very similar to that of a function, i.e. working like a “black-box”. In each iteration,  $a$  takes in a history  $d$  (the length of  $d$  depends on the iteration number) as an input and generates a solution  $x$  as the output, i.e.  $a(d) = x$ . The only difference is that  $a$  performs iteratively until the process terminates, while  $f$  does not. We can infer the nature of  $f$  by giving an input once at a time, but we can only explore that of  $a$  by generating a partial history  $d_{|X|-1} = [(x_1, y_1), \dots, (x_{|X|-1}, y_{|X|-1})]$ , which is formed through  $|X|-1$  function evaluations<sup>4</sup>. To summarize,  $a$  is a set of input-output mapping rules (logics) defined for any possible histories. Based on the previous definition of functions, for any  $f_i, f_j \in F$ ,  $f_i$  is equivalent to  $f_j$  if and only if  $f_i(x_k)$  is equal to  $f_j(x_k)$  for all  $k \in \{1, 2, \dots, |X|\}$ . Let  $A$  be the set of all search algorithms. Therefore, for any  $a_i, a_j \in A$ ,  $a_i$  is equivalent to  $a_j$  if and only if  $a_i(d_k)$  is equal to  $a_j(d_k)$  for all  $k \in \{1, 2, \dots, |X|-1\}$  and  $d_k \in D_k$ .

### 3. DETERMINISTIC ALGORIHTMS

We can have the complete picture about a function by enumeration, as in Table I. We can also do something similar on algorithms by enumerating all possible histories. This is possible, provided that the set of all possible complete histories  $D_{|X|}$  is finite.  $D_{|X|}$  is generated by enumerating all possible values from the sets  $X$  and  $Y$ , and we have

---

<sup>3</sup> All possible  $x$ -components of histories are just all permutations of  $X$ . Thus, the total number equals  $|X|! = 1 \times 2 \times \dots \times |X|$ .

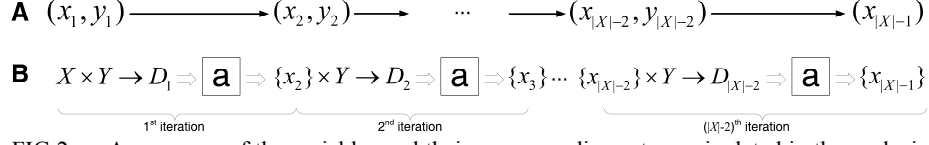


FIG 2. A summary of the variables and their corresponding sets manipulated in the analysis. (A) The sequence of variables which constitutes the history of an algorithm.  $y_{|X|-1}$ ,  $x_{|X|}$  and  $y_{|X|}$  are unimportant to the analysis, and thus, they are not shown here. (B) Iterations of an algorithm. In iteration  $m$  where  $m=1, \dots, |X|-1$ , the input to the algorithm  $a$  is a partial history  $d_m \in D_m$ , which is formed by concatenating the  $(x_m, y_m)$  pair to all the previously generated  $(x, y)$  pairs, i.e.  $d_{m-1}$ . Besides  $x_1, x_2, \dots, x_{|X|}$  are determined by the algorithm deterministically.

explained that  $X$  and  $Y$  implemented in computers must be finite. Therefore,  $D_{|X|}$  must be finite and be completely displayed in an enumeration table, e.g. Table III.

Consider a system with the sets  $X$  and  $Y$ , and we employ a deterministic algorithm  $a$  to solve an unknown function  $f$ . Recall that an algorithm must be well-defined in the sense that it should be able to generate an output with all possible inputs. The variables and their corresponding sets required in the analysis are shown in Figure 2. We begin with the 1<sup>st</sup> iteration. We input  $x_1$  from  $X$ . Note that this initial value  $x_1$  is picked by the user and independent of  $a$ . Thus,  $a$  needs to consider all elements in  $X$  as possibilities for  $x_1$ . Since  $f$  is unknown to us, all elements in  $Y$  are possible for  $y_1$ . Therefore, we pick  $(x_1, y_1)$  from the cross product of  $X$  and  $Y$ , i.e.,  $X \times Y$ . The number of possible  $(x_1, y_1)$  pairs is  $|X||Y|$ . There must be internal logics for  $a$  to handle all the possible  $(x_1, y_1)$ , and to select  $x_2$  for each of the possibilities. For  $x_2$ , we can only have  $|X|-1$  choices as one  $x$  in  $X$  has already been determined as  $x_1$ . We have *Lemma 1* for determining the total number of possible sets of deterministic rules (i.e. mapping) for the internal logics of  $a$ .

LEMMA 1. Consider a deterministic algorithm  $a$  which is a mapping from the set containing all possible  $d$  to the set containing all possible  $x$ , i.e.  $a(d) = x$ . If there are  $m$  and  $n$  possibilities for  $d$  and  $x$ , respectively, there will be  $n^m$  possible different  $a$ .

PROOF.  $a$  is a function with domain containing  $d$  and range containing  $x$ . There are  $n^m$  such possible functions. This proves the result.  $\square$

---

<sup>4</sup> From the algorithm's perspective, the  $(x_{|X|}, y_{|X|})$  pair is useless, as it will never be used to generate  $x_{|X|+1}$ .

As the numbers of possible  $d_1 = [(x_1, y_1)]$  and  $x_2$  are  $|X||Y|$  and  $|X|-1$ , respectively, by *Lemma 1*, there are  $(|X|-1)^{|X||Y|}$  different possible sets of rules for  $a$  for the 1<sup>st</sup> iteration. Assume the set of rules corresponding to  $a$  chooses  $x_2$  for each possible partial history  $d_1$  in  $D_1$ . Let us consider the 2<sup>nd</sup> iteration. Just as for  $y_1$ , there are  $|Y|$  possibilities for  $y_2$ . So there are  $|X||Y|^2$  possible  $d_2 = [(x_1, y_1), (x_2, y_2)]$  in  $D_2$  because  $x_2$  at this moment has been determined by  $a$ . We continue the analysis until the  $(|X|-2)$ <sup>th</sup> iteration. There are  $|X||Y|^{|X|-2}$  possible  $d_{|X|-2} = [(x_1, y_1), (x_2, y_2), \dots, (x_{|X|-2}, y_{|X|-2})]$  in  $D_{|X|-2}$ . For  $x_{|X|-1}$ , there are only two  $x$  which have not been chosen in the previous iterations. From *Lemma 1*, there are  $2^{|X||Y|^{|X|-2}}$  possible different sets of internal logics for the  $(|X|-2)$ <sup>th</sup> iteration. In the final iteration, i.e. the  $(|X|-1)$ <sup>th</sup> iteration,  $a$  does not actually use any internal logics to get the final  $x_{|X|}$ . It can simply get  $x_{|X|}$  by finding which  $x$  in  $X$  has not been included in the  $x$ -component of  $d_{|X|-1}$ . To differentiate one algorithm from another, this is useless. We can stop the analysis here and conclude with *Theorem 1*.

**THEOREM 1.** *Given a system with the sets  $X$  and  $Y$ , with  $|X| \geq 3$  and  $|Y| \geq 2$ , there*

*are  $\prod_{i=2}^{|X|-1} i^{|X||Y|^{|X|-i}}$  possible different deterministic algorithms.*

**PROOF.** It is trivial to see that the Theorem is meaningful only when  $|X|$  and  $|Y|$  are finite natural numbers. When  $|X| \leq 2$ , for any  $|Y|$ , no algorithms can be defined as all possible histories have been determined once we give the initial  $x_1$ . When  $|Y| = 1$ , there is only one possible function and we do not need any algorithms to solve it. Thus, we only need to prove the result for  $|X| \geq 3$  and  $|Y| \geq 2$ .

As shown in the previous analysis, there are  $(|X|-1)^{|X||Y|}$  different sets of internal logics in the 1<sup>st</sup> iteration. In the 2<sup>nd</sup> iteration, there are  $|X||Y|^2$  and  $(|X|-2)$  possibilities for  $d_2$  and  $x_3$ , respectively. Thus, there are  $(|X|-2)^{|X||Y|^2}$  different sets of internal logics, by *Lemma 1*. By applying the same reasoning for the rest of the iterations, we see that there are  $(|X|-3)^{|X||Y|^3}, \dots, (|X|-(|X|-2))^{|X||Y|^{|X|-2}}$  different sets for the 3<sup>rd</sup>, ...,  $(|X|-2)$ <sup>th</sup> iterations. A deterministic algorithm should have internal logics for all the iterations. When we consider all iterations together, the number of different deterministic algorithms equals

TABLE IV. Analysis of the example with  $X=\{1,2,3\}$  and  $Y=\{1,2,3\}$ .

Row number	$x_1$	$y_1$	Choices for $x_2$
1	1	1	{2,3}
2	1	2	{2,3}
3	1	3	{2,3}
4	2	1	{1,3}
5	2	2	{1,3}
6	2	3	{1,3}
7	3	1	{1,2}
8	3	2	{1,2}
9	3	3	{1,2}

TABLE V. Two deterministic algorithms for the example with  $X=\{1,2,3\}$  and  $Y=\{1,2,3\}$ .

Row number	$a_1$			$a_2$		
1	1	1	2	1	1	2
2	1	2	2	1	2	3
3	1	3	2	1	3	2
4	2	1	1	2	1	3
5	2	2	1	2	2	1
6	2	3	1	2	3	3
7	3	1	1	3	1	1
8	3	2	1	3	2	2
9	3	3	1	3	3	1

$$\begin{aligned}
 & (|X|-1)^{|X||Y|} \times (|X|-2)^{|X||Y|^2} \times \dots \times (|X|-(|X|-2))^{|X||Y|^{|X|-2}} \\
 &= (|X|-1)^{|X||Y|} \times (|X|-2)^{|X||Y|^2} \times \dots \times (2)^{|X||Y|^{|X|-2}} \\
 &= \prod_{i=2}^{|X|-1} i^{|X||Y|^{|X|-i}}.
 \end{aligned}$$

This completes the proof.  $\square$

### 3.1 An Example

We demonstrate the foregoing analysis with an example in which  $X=\{1,2,3\}$  and  $Y=\{1,2,3\}$ . Consider a system with  $X=\{1,2,3\}$  and  $Y=\{1,2,3\}$ . In the 1<sup>st</sup> iteration, there are  $|X||Y|=3 \times 3 = 9$  possibilities of  $d_1 = [(x_1, y_1)]$ . For  $x_2$ , there are two choices left. We list all possible partial histories  $d_1 = [(x_1, y_1)]$  and the choices for  $x_2$  in Table IV (where each row gives one possible  $d_1$ ). As there are at most  $|X|-1$  iterations, the longest history to define the internal logic of an algorithm is  $d_{|X|-2}$ , i.e.  $d_1$ , which has only the  $(x_1, y_1)$  pair. We can determine a particular algorithm by specifying the choices for  $x_2$  for all the partial histories. If we use “0” and “1” to indicate the 1<sup>st</sup> and the 2<sup>nd</sup> choices, respectively for

each partial history, we can specify an algorithm with a 9-digit sequence of “0” and “1”. For example, a sequence of “010101010” means that we choose “2” for the 1<sup>st</sup> row, “3” for the 2<sup>nd</sup>, “2” for the 3<sup>rd</sup>, and so on. Based on simple binary calculation, there are at most  $2^9$  corresponding numbers in decimal representation and each of these numbers can be used to represent a deterministic algorithm. We give two possible deterministic algorithms for the example in Table V.  $a_1$  corresponds to the algorithm in which we always select the first choice for  $x_2$  for each partial history, as in Table IV. We get  $a_2$  when selecting the 1<sup>st</sup> choice for odd-numbered rows and 2<sup>nd</sup> choice for the even-numbered rows.

With *Theorem 1*, we can further have *Corollary 1*.

**COROLLARY 1.** *Given a system with the sets  $X$  and  $Y$ , with  $|X| \geq 3$  and  $|Y| \geq 2$ , the set of all deterministic algorithms is finite. We can list all of them by indicating the choices of  $x_2, x_3, \dots$ , and  $x_{|X|-1}$  one-by-one for all the partial histories  $d_1, d_2, \dots$ , and  $d_{|X|-2}$ , respectively.*

**PROOF.** The proof can be directly deduced from *Theorem 1*, simply by listing all the possible sets of internal logics to determine  $x_2, x_3, \dots$ , and  $x_{|X|-1}$ . By selecting the choices sequentially for all possible partial histories  $d_1, d_2, \dots$ , and  $d_{|X|-2}$ , we can list all deterministic algorithms.  $\square$

#### 4. PROBABILISTIC ALGORITHMS

Deterministic and probabilistic algorithms together constitute all the algorithms. Probabilistic algorithms employ some randomness in their internal logics to generate outputs, while deterministic algorithms do not. Note that there are two kinds of randomness which allow us to get different results in different runs of the same algorithm. The first kind is due to the initial input  $x_1$ , which is not controlled by the algorithms. This randomness exists in both probabilistic and deterministic algorithms, and thus, both of them should have sufficient internal logics to handle it. The second kind is due to the internal logics of the algorithms, i.e. the algorithm may produce different outputs given the same history. This kind of randomness distinguishes probabilistic algorithms from deterministic ones.

TABLE VI. Different representations of a deterministic algorithm.

Enumerative form	Instructive form (lower-level)	Instructive form (higher-level)	Instructive form (highest-level)
1 1 2	if $(x_1, y_1)=(1,1)$ , then $x_2 \leftarrow -2$ ;	if $y_1=1$ , then $x_2 \leftarrow -x_1+1$ ;	if $y_1=1$ or 3, then $x_2 \leftarrow$
1 2 3	if $(x_1, y_1)=(1,2)$ , then $x_2 \leftarrow -3$ ;	if $y_1=2$ , then $x_2 \leftarrow -x_1-1$ ;	$x_1+1$ ;
1 3 2	if $(x_1, y_1)=(1,3)$ , then $x_2 \leftarrow -2$ ;	if $y_1=3$ , then $x_2 \leftarrow -x_1+1$ ;	if $y_1=2$ , then $x_2 \leftarrow -x_1-1$ ;
2 1 3	if $(x_1, y_1)=(2,1)$ , then $x_2 \leftarrow -3$ ;	if $x_2 > 3$ , then $x_2 \leftarrow -x_2-3$ ;	if $x_2 > 3$ , then $x_2 \leftarrow -x_2-3$ ;
2 2 1	if $(x_1, y_1)=(2,2)$ , then $x_2 \leftarrow -1$ ;		
2 3 3	if $(x_1, y_1)=(2,3)$ , then $x_2 \leftarrow -3$ ;		
3 1 1	if $(x_1, y_1)=(3,1)$ , then $x_2 \leftarrow -1$ ;		
3 2 2	if $(x_1, y_1)=(3,2)$ , then $x_2 \leftarrow -2$ ;		
3 3 1	if $(x_1, y_1)=(3,3)$ , then $x_2 \leftarrow -1$ ;		

We have shown that any deterministic algorithm can, in fact, be represented by a matrix of numbers (We call it the logic matrix of an algorithm, which is a mapping matrix defining the mapping logics. See Table V for examples). The numbers can just be the indices of their corresponding sets (as shown in Table V). They can be anything meaningful, e.g. real numbers, complex numbers, etc. The odd- and even-numbered columns of such matrices give the  $x$ - and  $y$ -components of the histories respectively, according to their orders of appearance in the histories. For example, the 1<sup>st</sup> column corresponds to the values of  $x_1$ , the 2<sup>nd</sup> to  $y_1$ , the 3<sup>rd</sup> to  $x_2$ , and so on. For given  $X$  and  $Y$ , deterministic algorithms should have identical columns for all the  $y$ . For example, for  $i = 1, \dots, |X|-2$ , the  $y_i$  column for algorithm  $a_1$  is identical to the  $y_i$  column of algorithm  $a_2$ . Note that it is unnecessary to enumerate the values for  $y_{|X|-1}$ ,  $x_{|X|}$ , and  $y_{|X|}$  because they are useless for understanding the internal logics behind the algorithm. Hence the corresponding columns are not included in the logic matrices. The rows reveal all possible distinct complete histories (with  $y_{|X|-1}$ ,  $x_{|X|}$  and  $y_{|X|}$  removed), which can be realized by that algorithm. In other words, we list all possible combinations of  $x_1$ ,  $y_1$ ,  $y_2, \dots$ , and  $y_{|X|-2}$ , as the rows. A deterministic algorithm explicitly tells which values of  $x_2$ ,  $x_3, \dots$ , and  $x_{|X|-1}$  are taken for all the rows. We define each row as a primitive logic. We can also interpret the logic matrix as a set of instructions by direct translation (see Table VI), and these instructions constitute the primitive form of the internal logics of a deterministic algorithm. In Table VI, The enumerative representation means the original matrix format. The 2<sup>nd</sup> column is just the direct translation of the 1<sup>st</sup> into more understandable instructions. We can further combine the redundancies and give a more compact version as in the 3<sup>rd</sup> column. An even higher-level representation is possible. In

this example, we give the highest-level representation in the 4<sup>th</sup> column. Therefore, we express the internal logics of an algorithm in terms of the logic matrix. Here we have *Lemma 2, Theorem 2, and Lemma 3.*

LEMMA 2. *Given a system with sets  $X$  and  $Y$ , with  $|X| \geq 3$  and  $|Y| \geq 2$ , any specific  $d_{|X|}^y$  can be determined exactly  $|X|$  times in the enumeration table by each possible deterministic algorithm.*

PROOF.  $|X| \geq 3$  and  $|Y| \geq 2$  ensure that we have proper definition of algorithms and meaningful functions. We have shown that a deterministic algorithm can be represented by a logic matrix, whose entries in every even column represent all possible  $y$ -values computable by that algorithm. The first even column, i.e. 2<sup>nd</sup> column, gives the values of  $y_1$ . The second even column, i.e. 4<sup>th</sup> column, gives the values of  $y_2$ , and so on. There are only  $(|X|-2)$  even columns as  $y_{|X|-1}$  and  $y_{|X|}$  are not needed to distinguish between algorithms.

We prove the result by mathematical induction. Let  $P(n)$  be the statement “any specific  $d_n^y$  can be determined by each possible deterministic algorithm exactly  $n$  times”. When  $n = 3$ , let  $d_3^y(i)$ , for  $1 \leq i \leq 3$ , represents the value of  $y_i$  specified by  $d_3^y$ . All deterministic algorithms have the same 2<sup>nd</sup> column in their logic matrices (see Table V for example). That means any value of  $d_3^y(1)$  can be determined by any algorithm. As there are no columns responsible for  $y_2$  and  $y_3$ , any  $d_3^y(2)$  and  $d_3^y(3)$  are possibly computed by every algorithm. Thus, any specific  $d_3^y$  can be determined by each possible algorithm. In the 2<sup>nd</sup> column of any possible logic matrix, the number of rows corresponding to any specific  $d_3^y(1)$  is three. Therefore,  $P(3)$  is true.

Assume that  $P(k)$ , where  $k \geq 3$ , is true. When  $n = k+1$ , the definition of algorithm guarantees that every possible logic matrix has identical 1<sup>st</sup> and 2<sup>nd</sup> columns. Consider all rows of a logic matrix for a particular value of  $x_1$ , all possible values  $y_1$  are included in these rows. All the even columns in these rows, except the 2<sup>nd</sup> column, are exactly those even columns of a logic matrix when  $n = k$ . In other words, a specific  $[d_{k+1}^y(2), \dots, d_{k+1}^y(k+1)]$  appears  $k$  times in these rows. When  $d_{k+1}^y(1)$  is also considered, a

specific  $d_{k+1}^y$  appears only once. Since there are  $(k+1)$  possible values of  $x_1$ ,  $P(k+1)$  is also true. By mathematical induction,  $P(n)$  is true for all  $n \geq 3$ .

The above analysis considers the occurrence of a particular  $d_{|X|}^y$  among all possible  $d_{|X|}^y$ , and the enumeration table is a representation of all possible  $d_{|X|}^y$ . Therefore, the result is valid for the enumeration table.

This completes the proof.

**THEOREM 2.** *For  $1 \leq i \leq |X|$ , let  $d_{|X|}^y(i)$  be the value of  $y_i$  specified in  $d_{|X|}^y$ . Given a system with sets  $X$  and  $Y$ , with  $|X| \geq 3$  and  $|Y| \geq 2$ , any specific  $[d_{|X|}^y(1), d_{|X|}^y(2), \dots, d_{|X|}^y(i)]$  can be determined the same number of times in the enumeration table by each possible deterministic algorithm.*

**PROOF.** As  $[d_{|X|}^y(i+1), \dots, d_{|X|}^y(|X|)]$  is not specified, we consider all possible combinations of  $y$ -value. As  $|Y|$  is a finite positive number, the number of possible combinations is also finite. For each combination attached to  $[d_{|X|}^y(1), d_{|X|}^y(2), \dots, d_{|X|}^y(i)]$ , by *Lemma 2*, each possible deterministic algorithm computes  $[d_{|X|}^y(1), d_{|X|}^y(2), \dots, d_{|X|}^y(|X|)]$  the same number of times. When we consider  $[d_{|X|}^y(1), d_{|X|}^y(2), \dots, d_{|X|}^y(i)]$  only, the result is also true.

**LEMMA 3.** *Deterministic algorithms are those which contain the least number of primitive logics.*

**PROOF.** If we follow *Corollary 1* to list all deterministic algorithms, we find that all of them are matrices of the same size. They all have the same number of columns, and this number is independent of the algorithms, and only depends on the set  $X$ . The rows are created just to handle possibilities raised from different combinations of  $x_1, y_1, y_2, \dots$ , and  $y_{|X|-2}$ , and all of them are also independent of the algorithms. Removal of a particular row means that the corresponding combination of  $x_1, y_1, y_2, \dots$ , and  $y_{|X|-2}$  cannot be handled by the algorithm. This violates the definition of algorithms because the resulting algorithm would not be well-defined anymore.  $\square$

As explained in the proof of *Lemma 3*, we cannot remove any rows from the enumerative form of deterministic algorithms. But we can add more. The effect is equivalent to creating more than one outcome for some particular partial history. This constitutes probabilistic algorithms. We can understand better with the examples below.

TABLE VII. Two examples demonstrating how to construct probabilistic algorithms from deterministic algorithms.

	Enumerative representation				Instructive representation
	Deterministic algorithms				
Example 1	$\begin{matrix} 1 & 1 & 2 \\ 1 & 2 & 3 \\ 1 & 3 & 2 \\ 2 & 1 & 3 \\ 2 & 2 & 1 \\ 2 & 3 & 3 \\ 3 & 1 & 1 \\ 3 & 2 & 2 \\ 3 & 3 & 1 \end{matrix}$	$\begin{matrix} 1 & 1 & 3 \\ 1 & 2 & 3 \\ 1 & 3 & 2 \\ 2 & 1 & 3 \\ 2 & 2 & 1 \\ 2 & 3 & 3 \\ 3 & 1 & 1 \\ 3 & 2 & 2 \\ 3 & 3 & 1 \end{matrix}$	$\begin{matrix} 1 & 1 & 2 \\ 1 & 1 & 3 \\ 1 & 2 & 3 \\ 1 & 3 & 2 \\ 2 & 1 & 3 \\ 2 & 2 & 1 \\ 2 & 3 & 3 \\ 3 & 1 & 1 \\ 3 & 2 & 2 \\ 3 & 3 & 1 \end{matrix}$	$\begin{matrix} 1 & 1 & 2 \\ 1 & 1 & 3 \\ 1 & 2 & 3 \\ 1 & 3 & 2 \\ 2 & 1 & 3 \\ 2 & 2 & 1 \\ 2 & 3 & 3 \\ 3 & 1 & 1 \\ 3 & 2 & 2 \\ 3 & 3 & 1 \end{matrix}$	<p>if <math>y_1 = 1</math>, then {  if <math>x_1 = 1</math>, then <math>x_2 \leftarrow 2</math> or <math>3</math> ;  else <math>x_2 \leftarrow x_1 + 1</math> ;  };  if <math>y_1 = 2</math>, then <math>x_2 \leftarrow x_1 - 1</math> ;  if <math>y_1 = 3</math>, then <math>x_2 \leftarrow x_1 + 1</math> ;  if <math>x_2 &gt; 3</math>, then <math>x_2 := x_2 - 3</math> ;</p>
Example 2	$\begin{matrix} 1 & 1 & 2 \\ 1 & 2 & 3 \\ 1 & 3 & 2 \\ 2 & 1 & 3 \\ 2 & 2 & 1 \\ 2 & 3 & 3 \\ 3 & 1 & 1 \\ 3 & 2 & 2 \\ 3 & 3 & 1 \end{matrix}$	$\begin{matrix} 1 & 1 & 3 \\ 1 & 2 & 3 \\ 1 & 3 & 2 \\ 2 & 1 & 3 \\ 2 & 2 & 1 \\ 2 & 3 & 3 \\ 3 & 1 & 1 \\ 3 & 2 & 2 \\ 3 & 3 & 1 \end{matrix}$	$\begin{matrix} 1 & 1 & 2 \\ 1 & 2 & 3 \\ 1 & 3 & 2 \\ 2 & 1 & 1 \\ 2 & 2 & 1 \\ 2 & 3 & 3 \\ 3 & 1 & 1 \\ 3 & 2 & 2 \\ 3 & 3 & 1 \end{matrix}$	$\begin{matrix} 1 & 1 & 3 \\ 1 & 2 & 3 \\ 1 & 3 & 2 \\ 2 & 1 & 1 \\ 2 & 2 & 1 \\ 2 & 3 & 3 \\ 3 & 1 & 1 \\ 3 & 2 & 2 \\ 3 & 3 & 1 \end{matrix}$	<p>if <math>y_1 = 1</math>, then  <math>x_2 \leftarrow \text{any } x \in \{1, 2, 3\} \setminus \{x_1\}</math> ;  if <math>y_1 = 2</math>, then <math>x_2 \leftarrow x_1 - 1</math> ;  if <math>y_1 = 3</math>, then <math>x_2 \leftarrow x_1 + 1</math> ;  if <math>x_2 &gt; 3</math>, then <math>x_2 \leftarrow x_2 - 3</math> ;</p>

#### 4.1 Examples

We give two examples in Table VII. In example 1, we make the first given deterministic algorithm as the starting point. We try to produce a new probabilistic algorithm by giving one more variation to  $x_2$  of the primitive logic [1 1 2] of the base. It is equivalent to adding one more primitive logic [1 1 3] to it. The effect is equal to combining the two given deterministic algorithms with all the redundancies removed.

In example 2, we work on the same starting algorithm and try to give one more variation to  $x_2$  of each of the primitive logics [1 1 2], [2 1 3], and [3 1 1]. It is equivalent

to adding [1 1 3], [2 1 1] and [3 1 2] to the base. The effect is equivalent to combining the eight given deterministic algorithms with all the redundancies removed.

We can further have *Theorem 3* and *Corollary 3*.

**THEOREM 3.** *Given a system with the sets  $X$  and  $Y$ , with  $|X| \geq 3$  and  $|Y| \geq 2$ , the corresponding deterministic algorithms form the basis of its algorithmic space, which consists of all deterministic and probabilistic algorithms. All algorithms are formed by probabilistically combining some of the deterministic algorithms.*

**PROOF.** Algorithms are characterized by internal logics which are expressed as a set of primitive logics. Thus, all algorithms can be represented in the enumerative form, i.e. logic matrix. From *Lemma 3*, algorithms cannot be made with a number of primitive logics less than that of the deterministic algorithms. Thus, probabilistic algorithms must have more primitive logics than deterministic algorithms. This is tantamount to adding more primitive logics to deterministic algorithms. Now the question becomes whether we can add an arbitrary number of primitive logics or not. Here, we are going to prove the answer is negative.

A primitive logic is composed of values for  $x_1, y_1, x_2, y_2, \dots, x_{|X|-1}$ . An algorithm must contain a sufficient number of primitive logics for all possible variations of  $x_1, y_1, y_2, \dots, y_{|X|-2}$ , in order to make the algorithm well-defined. A probabilistic algorithm is just one which has some primitive logics to allow only some of the variations of  $x_2, x_3, \dots, x_{|X|-1}$ . Suppose we have a deterministic algorithm, which acts as the base, and pick a primitive logic from it. We decide to give one more possible value to  $x_i$  only, where  $2 \leq i \leq |X| - 1$ . First we produce a new primitive logic by replicating all its previous  $x_1, y_1, x_2, y_2, \dots$ , up to  $y_{i-1}$  from the originally chosen primitive logic, because we are not interested in making variations to them. Then we assign the value of the additional variation to  $x_i$ . Next we need to add sufficiently more primitive logics (with all  $x_1, y_1, x_2, y_2, \dots$ , up to  $x_i$  identical) to account for the variations raised from  $y_i, y_{i+1}, \dots, y_{|X|-2}$ . Otherwise, the resulting probabilistic algorithm is not well-defined. For  $x_{i+1}, x_{i+2}, \dots, x_{|X|-1}$ , we can try our best to follow their original values from the chosen primitive logic, but we cannot repeat the new value of  $x_i$  in any of  $x_{i+1}, x_{i+2}, \dots, x_{|X|-1}$ . For example, we have  $X = \{1, 2, 3, 4, 5, 6\}$  and the  $x$ -components of the chosen primitive logic are  $[x_1, x_2, x_3, x_4, x_5] = [1, 2, 3, 4, 6]$ . We decide

to give one more variation to  $x_3$ , the only possibilities are  $\{4, 5, 6\}$ . If we take “5”, the  $x$ -components of the new primitive logic will become  $[1, 2, 5, 4, 6]$ . But if we take “4” (or “6”) as  $x_3$ , we cannot duplicate it for  $x_4$  (or  $x_5$ ). We can simply make them as  $[1, 2, 4, 3, 6]$  (or  $[1, 2, 6, 4, 3]$ ). This is similar to interchanging the positions assigned with the two variations of  $x_i$  in the original primitive logic. Thus, we need to add a group of new primitive logics for one variation given to  $x_i$ . Clearly, we must be able to construct another deterministic algorithm with this group of new primitive logics, together with certain primitive logics of the base. Therefore, the new probabilistic algorithm created by giving one more variation to  $x_i$  is actually composed of two deterministic algorithms.

Moreover, we can create many more probabilistic algorithms, just due to two possible values of  $x_i$ , by giving different probability distributions to the possible values of  $x_i$ .

If we decide to give more variations to the same  $x_i$ , or even to another  $x_i$ , we just follow the above steps to create a group of primitive logics for each variation at a time. All the deterministic algorithms can be specified probabilistically in a probabilistic distribution.

This completes the proof.  $\square$

**COROLLARY 3.** *Given a system with the sets  $X$  and  $Y$ , with  $|X| \geq 3$  and  $|Y| \geq 2$ , an algorithm, both deterministic and probabilistic, can be represented by a probabilistic distribution on the set of the deterministic algorithms.*

**PROOF.** This can be directly derived from *Theorem 3*.  $\square$

We discover that algorithms are composed of deterministic algorithms. This nature is not readily detected due to our practice of implementing algorithms in computer as program codes. In Table VI, a deterministic algorithm can be represented by a logic matrix, as well as in its direct translation, i.e. the lower-level instructive form. By removing some redundancies, we can represent the same algorithm with different notations at higher levels. This is also true for probabilistic algorithms (see Table VII). In normal practice, this is exactly what we are doing when writing program codes to represent algorithms in the computer, i.e. we often compile algorithms in high-level languages, instead of assembly language or machine code. When comparing the size, we can write an algorithm in high-level languages with much fewer lines of code than that in

the low-level ones. Thus, we prefer high-level languages most of the time. We can see that a single line of instruction at a higher level is composed of components of a vast number of deterministic algorithms. Moreover, the sets of  $X$  and  $Y$  are large in practice, and so is the total number of definable deterministic algorithms. In other words, a probabilistic algorithm in high level form, which is what we often see, can be represented by even more deterministic algorithms. The way we compile algorithms prevent us from revealing the nature of search algorithms.

## 5. MATCHING

Understanding the nature of search algorithms is useful. We demonstrate one of its applications as follows.

It is generally accepted that a search algorithm can work well only when matched with the right problem type. To the best of our knowledge, it is just an intuition and no one has actually proved or disproved this statement. Using our discoveries regarding the nature of search algorithms, we can show that it is indeed correct.

Recall that  $F$  is the set of all possible functions.  $F$  is finite and we can list all of them, e.g. Table I. The same type of functions (or problems) will share some common characteristics. One may be tempted to divide  $F$  into different types according to characteristics identifiable in the “table of functions” such as Table I. For example, we may divide the functions listed in Table I into three types according to the values of  $f(3)$ , but this division may not correspond to any meaningful high-level characterization of the functions.

To understand this matching, we can analyze the enumeration table. Consider the same example again and we have all possible histories generated by all algorithms in the enumeration table (see Table III). First of all, we need to define the performance metric. For example, suppose an algorithm is considered good if it can generate a history whose  $y_2$  is the minimum of the function. We can find all good histories in the table. Then we may characterize an algorithm by the set of all possible histories it can generate. We demonstrate the above process with Table VIII and Table IX for performance of  $a_1$  from Table V and that of  $a_2$  from Table V, respectively. Here performance is evaluated by indicating when the minimum  $y$ -value is located. In these examples shown in Table VIII and Table IX, histories are of the best performance when  $y_2$  is minimum. We do not consider  $y_1$  because  $x_1$  is not determined by the algorithms. Those histories of the best performance are highlighted in blue, while those determined by the corresponding

algorithm are highlighted in yellow. Those which are mixed with blue and yellow are changed to green. For each of the tables, the numbers of histories in green for all the functions are summarized in the last column. These numbers reveal the performance of the algorithms on particular functions. An algorithm is of superior performance if it can generate more histories of good quality. Consider deterministic algorithms first. For a particular function or function type (assume we can identify a function type in the “table of function”), we can tailor an algorithm by specifying the desired histories as much as possible. The one with the largest number of desired histories is the best match. We can also say that a probabilistic algorithm is matched with a function or function type if it includes the best-matched deterministic algorithm with a large probability in its probabilistic distribution. Then we have *Theorem 4*.

**THEOREM 4.** *For a particular function or function type, there exists a corresponding best-matched algorithm.*

**PROOF.** We can build a deterministic algorithm by selecting desired histories in the enumeration table. To construct a probabilistic algorithm to match a particular problem, we can include the deterministic algorithm with the most desirable set of histories and assign it a high probability value. This proves the result.  $\square$

TABLE VIII. Performance of algorithms for the example where  $X=\{1,2,3\}$  and  $Y=\{1,2,3\}$ .

Table entries are y-components of the histories	x-components of the histories						Number of histories in green
	3 2 1	3 1 2	2 3 1	2 1 3	1 2 3	1 3 2	
$f_1$	1 1 1	1 1 1	1 1 1	1 1 1	1 1 1	1 1 1	3
$f_2$	1 1 2	1 2 1	1 1 2	1 2 1	2 1 1	2 1 1	1
$f_3$	1 1 3	1 3 1	1 1 3	1 3 1	3 1 1	3 1 1	1
$f_4$	1 2 1	1 1 2	2 1 1	2 1 1	1 2 1	1 1 2	2
$f_5$	1 2 2	1 2 2	2 1 2	2 2 1	2 2 1	2 1 2	0
$f_6$	1 2 3	1 3 2	2 1 3	2 3 1	3 2 1	3 1 2	0
$f_7$	1 3 1	1 1 3	3 1 1	3 1 1	1 3 1	1 1 3	2
$f_8$	1 3 2	1 2 3	3 1 2	3 2 1	2 3 1	2 1 3	0
$f_9$	1 3 3	1 3 3	3 1 3	3 3 1	3 3 1	3 1 3	0
$f_{10}$	2 1 1	2 1 1	1 2 1	1 1 2	1 1 2	1 2 1	3
$f_{11}$	2 1 2	2 2 1	1 2 2	1 2 2	2 1 2	2 2 1	1
$f_{12}$	2 1 3	2 3 1	1 2 3	1 3 2	3 1 2	3 2 1	1
$f_{13}$	2 2 1	2 1 2	2 2 1	2 1 2	1 2 2	1 2 2	2
$f_{14}$	2 2 2	2 2 2	2 2 2	2 2 2	2 2 2	2 2 2	3
$f_{15}$	2 2 3	2 3 2	2 2 3	2 3 2	3 2 2	3 2 2	1
$f_{16}$	2 3 1	2 1 3	3 2 1	3 1 2	1 3 2	1 2 3	2
$f_{17}$	2 3 2	2 2 3	3 2 2	3 2 2	2 3 2	2 2 3	2
$f_{18}$	2 3 3	2 3 3	3 2 3	3 3 2	3 3 2	3 2 3	0
$f_{19}$	3 1 1	3 1 1	1 3 1	1 1 3	1 1 3	1 3 1	3
$f_{20}$	3 1 2	3 2 1	1 3 2	1 2 3	2 1 3	2 3 1	1
$f_{21}$	3 1 3	3 3 1	1 3 3	1 3 3	3 1 3	3 3 1	1
$f_{22}$	3 2 1	3 1 2	2 3 1	2 1 3	1 2 3	1 3 2	2
$f_{23}$	3 2 2	3 2 2	2 3 2	2 2 3	2 2 3	2 3 2	3
$f_{24}$	3 2 3	3 3 2	2 3 3	2 3 3	3 2 3	3 3 2	1
$f_{25}$	3 3 1	3 1 3	3 3 1	3 1 3	1 3 3	1 3 3	2
$f_{26}$	3 3 2	3 2 3	3 3 2	3 2 3	2 3 3	2 3 3	2
$f_{27}$	3 3 3	3 3 3	3 3 3	3 3 3	3 3 3	3 3 3	3

TABLE IX. Performance of algorithms for the example where  $X=\{1,2,3\}$  and  $Y=\{1,2,3\}$ .

Table entries are y-components of the histories	x-components of the histories						Number of histories in green
	3 2 1	3 1 2	2 3 1	2 1 3	1 2 3	1 3 2	
$f_1$	1 1 1	1 1 1	1 1 1	1 1 1	1 1 1	1 1 1	3
$f_2$	1 1 2	1 2 1	1 1 2	1 2 1	2 1 1	2 1 1	2
$f_3$	1 1 3	1 3 1	1 1 3	1 3 1	3 1 1	3 1 1	2
$f_4$	1 2 1	1 1 2	2 1 1	2 1 1	1 2 1	1 1 2	2
$f_5$	1 2 2	1 2 2	2 1 2	2 2 1	2 2 1	2 1 2	1
$f_6$	1 2 3	1 3 2	2 1 3	2 3 1	3 2 1	3 1 2	0
$f_7$	1 3 1	1 1 3	3 1 1	3 1 1	1 3 1	1 1 3	2
$f_8$	1 3 2	1 2 3	3 1 2	3 2 1	2 3 1	2 1 3	2
$f_9$	1 3 3	1 3 3	3 1 3	3 3 1	3 3 1	3 1 3	1
$f_{10}$	2 1 1	2 1 1	1 2 1	1 1 2	1 1 2	1 2 1	2
$f_{11}$	2 1 2	2 2 1	1 2 2	1 2 2	2 1 2	2 2 1	1
$f_{12}$	2 1 3	2 3 1	1 2 3	1 3 2	3 1 2	3 2 1	2
$f_{13}$	2 2 1	2 1 2	2 2 1	2 1 2	1 2 2	1 2 2	1
$f_{14}$	2 2 2	2 2 2	2 2 2	2 2 2	2 2 2	2 2 2	3
$f_{15}$	2 2 3	2 3 2	2 2 3	2 3 2	3 2 2	3 2 2	2
$f_{16}$	2 3 1	2 1 3	3 2 1	3 1 2	1 3 2	1 2 3	0
$f_{17}$	2 3 2	2 2 3	3 2 2	3 2 2	2 3 2	2 2 3	2
$f_{18}$	2 3 3	2 3 3	3 2 3	3 3 2	3 3 2	3 2 3	1
$f_{19}$	3 1 1	3 1 1	1 3 1	1 1 3	1 1 3	1 3 1	2
$f_{20}$	3 1 2	3 2 1	1 3 2	1 2 3	2 1 3	2 3 1	0
$f_{21}$	3 1 3	3 3 1	1 3 3	1 3 3	3 1 3	3 3 1	1
$f_{22}$	3 2 1	3 1 2	2 3 1	2 1 3	1 2 3	1 3 2	2
$f_{23}$	3 2 2	3 2 2	2 3 2	2 2 3	2 2 3	2 3 2	2
$f_{24}$	3 2 3	3 3 2	2 3 3	2 3 3	3 2 3	3 3 2	1
$f_{25}$	3 3 1	3 1 3	3 3 1	3 1 3	1 3 3	1 3 3	1
$f_{26}$	3 3 2	3 2 3	3 3 2	3 2 3	2 3 3	2 3 3	1
$f_{27}$	3 3 3	3 3 3	3 3 3	3 3 3	3 3 3	3 3 3	3

From the above discussion, we can see that a tailor-made deterministic algorithm must have better performance than any other probabilistic ones as the latter contain other non-tailor-made deterministic algorithm(s) in some portion of its probability distribution. The non-tailor-made portion may divert the algorithm to some unfavorable histories. However, it may not be possible to tailor an algorithm without “knowing” the function. Suppose we have a deterministic algorithm to solve a function. It can perform well only if it begins on and follows the right track, i.e. the algorithm can generate the desired histories before it is stopped. As the size of the enumeration table of a system used to solve standard problems is huge and a deterministic algorithm can only cover a tiny part, the chance of having the algorithm off-track is extremely high. However, a probabilistic

algorithm can cover more histories and the probability of getting off-track becomes lower. But this sacrifices efficiency. This kind of accuracy-efficiency tradeoff highly depends on our knowledge of the functions we are solving. Most of the time, it is advisable to employ a general-purpose algorithm, i.e. a probabilistic algorithm which covers many histories in the enumeration table, if we know very little or nothing about the function. However, we can engage a heuristic-type algorithm, i.e. a probabilistic algorithm which covers relatively smaller portions of the histories in the enumeration table, or even a deterministic algorithm, if we already have certain knowledge about the target function. This tradeoff between general-purpose algorithms and heuristics on solving search problems has been accepted in the research community. The nature of search algorithms provides the explanation.

## 6. CONCLUSION

We conclude this article with an analogy with atomic theory, which states that all matter in the world is composed of some basic units, i.e. atoms. Those atoms of the same electronic structure are grouped together as an element in the periodic table, e.g. hydrogen, carbon, sodium, etc. More complex substances are formed through various combinations of atoms from different elements. We can describe a chemical reaction, which changes reactants to products, in terms of the elements involved. This allows us to predict what the outputs of a reaction will be given the structures of reactants. We can also control the quality of the products by changing different portions of reactants. This is known as chemical synthesis. We can take wine fermentation as an example. Fermentation is a biochemical process which converts sugars into ethanol and carbon dioxide. It is one of the most important steps in winemaking and has been continuously refined since early human history. Before the underlying chemical reactions were well understood, it was easy to have stuck fermentation and wine faults due to inappropriate temperature, speed of fermentation, and level of oxygen. People tried to control and improve the process mainly through experience, observation, and by trial-and-error. After atomic theory has been developed, we are able to understand the connection between the components involved, to take control of the whole process, and to create various flavors and aromas of wine. Modern chemistry has been flourishing due to atomic theory.

This work is dedicated to understanding the nature of search algorithms. We have shown that all search algorithms are composed of some basic units, namely, deterministic algorithms, whose total number can be determined. We can also list all of them and any

search algorithm can be represented as a probabilistic combination of the set of the defined deterministic algorithms. Our result allows us to better understand the connection between algorithms and the problems we are solving. We now understand why search problems can be matched with suitable algorithms to gain better performance. Given a problem, our result helps in choosing an appropriate algorithm and to predict the performance. We do not need to select an algorithm solely through experience and by trial-and-error. We believe our discovery on the nature of search algorithms allows us to have a better understanding of searching and optimization.

## REFERENCES

- CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. 2001. *Introduction to Algorithms*. 2nd. MIT Press, Cambridge, MA.
- KIRKPATRICK, S., GELATT, C.D., Jr, AND VECCHI M.P. 1983. Optimization by simulated annealing. *Science* 220, 671-680.
- WOLPERT, D.H., AND MACREADY, W.G. 1995. No free lunch theorems for search. Technical Report SFI-TR-95-02-010, Santa Fe Institute, 1399 Hyde Park Road, Santa Fe, NM.
- WOLPERT, D.H., AND MACREADY, W.G. 1997. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation* 1, 67-82.