National Conference on Theoretical Computer Science (NCTCS2015)

On Four Non-Computable Decision Problems ⋆

Kun Wang*, Nan Wu, Fangmin Song**

State Key Laboratory for Novel Software Technology,
Department of Computer Science and Technology,
Nanjing University, Nanjing 210023, China

Abstract

In this paper, we restudy busy beaver problem that is defined by Tibor Rado in 1962. Based on his work, we describe four decision problems — *busy beaver problem*, *max shifter problem*, *halting empty tape problem*, *halting problem* — and prove the Turing equivalence of them by reducing one problem to another under Turing computability. It has been proved that halting problem is non-computable, so we show that all these four problems are non-computable.

Keywords: Turing machine; Turing computability; Turing reduction; Halting problem; Busy beaver

1 Introduction

Some classical and interesting non-computable problems in computability theory are raised by Tibor Rado in 1962[1]. He considered Turing machines of n states which must conform to some restrictions, and asked following two questions:

- **busy beaver problem** When such a n-state machine is run on an empty tape, what is the largest number of 1's it may leave on the tape when it halts?
- **max shift problem** When such a n-state machine is run on an empty tape, what is the largest number of shifts it may take before it halts?

Based on these two non-computable problems, we describe some similar problems which can be represented by unary predicate functions. More accurately, we describe *busy beaver problem* (slightly different from the one defined by Rado), *max shifter problem* (slightly different from the one defined by Rado also), *halting empty tape problem*, and *halting problem*. Then we prove all of these problems are non-computable. As some of these problems are not that easy to figure

⋆Project supported by the Chinese National Natural Science Foundation(No. 61300050, No. 91321312, No. 61021062), the Chinese National Natural Science Foundation of Innovation Team(No. 60721002) and the Ph.D. Programs Foundation of Ministry of Education of China(No. 20120091120008).

*Graduate student of Department of Computer Science and Technology, Nanjing University.

**Corresponding author.

Email address: fmsong@nju.edu.cn (Fangmin Song).

out their non-computability directly, we apply a powerful mathematical tool — Turing reduction — to achieve our task. We proved that the four problems are equivalent, i.e., if there exists a program that can solve any one problem of them, the remaining three are easy to solve using this program. Halting problem is so well-known that there exists easy ways to show its non-computability, thus we complete our proof that all of these four problems are non-computable. We assume the readers are familiar with Turing machine and its variants, details on this topic can be found in [2][3][4][5][6] and references therein.

The paper is structured as follows. In Section 2, we give some preliminary knowledge. As there are a lot of terminologies, we only give concise definitions, more details can be found in bibliography. In Section 3, we describe the problems we want to solve in this paper, i.e., we define formally four predicates. Then we prove their equivalence in Section 4 using reduction. In the final section, we summarize the work and give a simple note on what still needs to be done.

2 Preliminaries

Before investigating the problems, we shall recall some terminologies in computing theory.

2.1 Computable Functions

Terminologies described in this section can be found in [7].

Partial Function. A *partial function on a set* S is simply a function whose domain is a subset of S. If f is a partial function on set S and $a \in S$, then we write

- $f(a)\downarrow$ and says that $f(a)$ is *defined* to indicate that a is in the domain of f;
- $f(a)\uparrow$ and says that $f(a)$ is *undefined* to indicate that a is not in the domain of f.

Total Function. If a partial function on set S has the domain S, then it is called *total*.

Partially Computable Function. For any program $\mathcal{P}$ with m inputs $x_1, \cdots, x_m$, the function $\Psi_{\mathcal{P}}^{(m)}(x_1, \cdots, x_m)$ is said to be computed by $\mathcal{P}$. In general, the program accepts one or more integers as input and generates one integer as output. A given partial function g is said to be *partially computable* if it is computed by some program. That is, g is partial computable if there is a program $\mathcal{P}$ such that $g(r_1, \cdots, r_m) = \psi_{\mathcal{P}}^{(m)}(r_1, \cdots, r_m)$ for all $r_1, \cdots, r_m$.

Computable Function. A function is said to be *computable* if it is both partial computable and total.

2.2 Predicate function

A Predicate is a special form of function, by a predicate of a Boolean-valued function on a set $\mathcal{S}$ we mean a total function $\mathcal{P}$ on $\mathcal{S}$ such that for each $a \in \mathcal{S}$, either $\mathcal{P}(a)$ = TRUE or $\mathcal{P}(a)$ = FALSE, where $TRUE$ and $FALSE$ are distinct objects called *truth values*[8]. For the purpose of coding programs by numbers, it is meaningful to identify the truth values with specific numbers TRUE = 1, FALSE = 0. In the context of computing theory, predicates are often

described as decision problems. If a problem can be formalized by a predicate, then the problem is computable is equal the predicate is computable.

There is close relationship between predicates and sets, so we will use the language of sets in proofing the equivalence of some problems. For example, the predicate $P(x)$ is the characteristic function of the set $\{x \in N|P(x)=1\}$. To say that a set S, where $S \in N^m$, belongs to some class of functions means that the characteristic function $P(x_1, \cdots, x_m)$ of S belongs to the class in question. The close relationship between predicates and sets is such that one readily translate back and forth between discourse involving one of these notions and discourse involving the other[7].

2.3 Reducibility

Reducibility showed up as new technique while proving whether or not a Turing machine halts for all inputs is unsolvable. It has traditionally been a very powerful and widely used tool in theoretical computer science as well as mathematics[9].

Reducibility always involves two problems, say $\mathcal{P}_1$ and $\mathcal{P}_2$. If $\mathcal{P}_1$ reduces to $\mathcal{P}_2$, then a solution for $\mathcal{P}_2$ can be used to solve $\mathcal{P}_1$. The reducibility says nothing about the actual solving of problems $\mathcal{P}_1$ and $\mathcal{P}_2$. It only deals with the solvability of problem $\mathcal{P}_1$, given a solution of problem $\mathcal{P}_2$[10].

Definition 1 *The set $\mathscr{A}$ is reducible to the set $\mathscr{B}$ (written $\mathscr{A} \preceq_t \mathscr{B}$) if and only if there is a totally computable function f such that for all x, $x \in \mathscr{A}$ if and only if $f(x) \in \mathscr{B}$.*

If we have an algorithm to convert instances of a problem $\mathscr{A}$ into instances of a problem $\mathscr{B}$ that have the same answer, then we say $\mathscr{A}$ reduces to $\mathscr{B}$. We can use this proof to show that $\mathscr{B}$ is at least as hard as $\mathscr{A}$[11]. Formally, a reduction from $\mathscr{A}$ to $\mathscr{B}$ is a Turing machine that takes an instance of $\mathscr{A}$written on its tape and halts with an instance of $\mathscr{B}$ on its tape. In practice, we shall generally describe reductions as if they were computer programs that take an instance of $\mathscr{A}$ as input and produce an instance of $\mathscr{B}$ as output. *The equivalence of Turing machines and computer programs allows us to describe the reduction by either means.*

Theorem 1 *If there is a reduction from $\mathscr{A}$ to $\mathscr{B}$, i.e., $\mathscr{A} \preceq_t \mathscr{B}$, then:*

1. *If $\mathscr{A}$ is non-computable, so is $\mathscr{B}$.*

2. *If $\mathscr{B}$ is computable, so is $\mathscr{A}$.*

Theorem 2 *If $\mathscr{A} \preceq_t \mathscr{B}$ and $\mathscr{B} \preceq_t \mathscr{C}$, then $\mathscr{A} \preceq_t \mathscr{C}$.*

Definition 2 *$\mathscr{A} \equiv_t \mathscr{B}$ means that $\mathscr{A} \preceq_t \mathscr{B}$ and $\mathscr{B} \preceq_t \mathscr{A}$.*

In general, for sets $\mathscr{A}$ and $\mathscr{B}$, if $\mathscr{A} \equiv_t \mathscr{B}$ then $\mathscr{A}$ and $\mathscr{B}$ are of the same difficulty. We say $\mathscr{A}$ and $\mathscr{B}$ are equivalent under Turing reduction and Turing computability.

2.4 Useful functions

Now we define some functions that are used in following sections. Each of these functions accomplishes a simple task and its Turing computability is easy to prove.

Pairing Function. Pairing function is a convenient coding device that use primitive recursive functions to code a pair of numbers by a single number[5]. Suppose $x, y \in \mathcal{N}$ are arbitrary numbers, the pairing function is defined as $< x, y >= 2^x(2y+1) - 1$. If $z \in \mathcal{N}$ is any given number, there is a *unique* solution x, y to the equation $< x, y >= z$. This equation defines functions $x = l(z), y = r(z)$. The definition of $l(z), r(z)$ can be expressed as $< x, y >= z$ if and only if $x = l(z)$ and $y = r(z)$.

Encoding Function. Let $\mathcal{M}$ denotes an arbitrary Turing machine, then $\rho(\mathcal{M})$ is the encoding number of $\mathcal{M}$.

State Counting Function. Let $\mathcal{M}$ denotes an arbitrary Turing machine, then $\delta(\mathcal{M})$ is the number of states $\mathcal{M}$ has. If $\mathcal{M}$ is encoded into a number x, $\delta(x)$ has the same meaning as $\delta(\mathcal{M})$.

Count Ones Function. Let x denotes an arbitrary Turing machine $\mathcal{M}$, then $CountOnes(x)$ is the number of 1s that $\mathcal{M}$ writes in the tape after halting when started on an empty tape.

Count Shifts Function. Let x denotes an arbitrary Turing machine $\mathcal{M}$, then $CountShifts(x)$ is the number of shifts that $\mathcal{M}$ takes before halting when started on an empty tape.

The Set of n-state Turing machines. Let $\Omega(n)$ denotes the set of all n-state Turing machines. There are exactly $\Omega(n) = (2 \cdot 2 \cdot (n+1))^{2n} = (4n+4)^{2n}$ different sets of instructions for n-state Turing machines[12].

The Set of n-state Turing machines Halting on Empty Tape. Let $\omega(n)$ define the set of all n-state Turing machines which eventually halt when started on a blank tape. It is trivial that $\omega(n) \subseteq \Omega(n)$.

Take One Step Function. Let x denotes an arbitrary Turing machine $\mathcal{M}$, $TakeOneStep(x)$ will simulate $\mathcal{M}$ and take just one transition of its states. The key importance of this function is that after taking one step, the function will record current configuration of Turing machine $\mathcal{M}$.

Get Current State Function. Let x denotes an arbitrary Turing machine $\mathcal{M}$, $GetCurrentState(x)$ denotes current state of the machine. We will use functions $TakeOneStep(x)$ and $GetCurrentState(x)$ together to get consecutive states of the given machine.

3 Description of Four Decision Problems

3.1 Busy beaver problem

The original busy beaver problem defined by Rado in [1] is a number-theoretic function, its relationship with the halting problem is not easy to figure out. We describe the *busy beaver problem* as: Given an arbitrary Turing machine $\mathcal{M}$, is it a busy beaver of $\delta(\mathcal{M})$ states?

Definition 3 *Predicate $\mathscr{O}$ for the busy beaver problem*

$$\mathscr{O}(x) = \begin{cases} 1, & \textit{Turing machine } \mathcal{M} \textit{ encoded by } x \textit{ is a busy beaver of } \delta(\mathcal{M}) \textit{ states;} \\ 0, & \textit{otherwise.} \end{cases}$$

With $\mathscr{O}$, it is easy to calculate the value of $\Sigma(n)$ ($n = \delta(\mathcal{M})$). The set of all Turing machines with n states is finite, thus we can test each $\mathcal{M}$ of them using $\mathscr{O}(\rho(\mathcal{M}))$. If $\mathcal{M}_i$ is a n-state busy beaver, then run $\mathcal{M}_i$ again to get the value of $\Sigma(n)$. By this informal reasoning, we show $\mathscr{O}$ is at least as hard as Σ.

3.2 Max shifter problem

The original max shifting problem defined by Rado is also a number-theoretic function. We describe the *max shifter problem* as: Given an arbitrary Turing machine $\mathcal{M}$, is it a max shifter of $\delta(\mathcal{M})$ states?

Definition 4 *Predicate $\mathscr{S}$ for the max shifter problem*

$$\mathscr{S}(x) = \begin{cases} 1, & \text{Turing machine } \mathcal{M} \text{ encoded by } x \text{ is a max shifter of } \delta(\mathcal{M}) \text{ states;} \\ 0, & \text{otherwise.} \end{cases}$$

With $\mathscr{S}$, it is easy to calculate the value of $S(n)$ ($n = \delta(\mathcal{M})$). The set of all Turing machines with n states is finite, thus we can test each $\mathcal{M}$ of them using $\mathscr{S}(\rho(\mathcal{M}))$. If $\mathcal{M}_j$ is a n-state max shifter, run $\mathcal{M}_j$ again to get the value of $S(n)$.

3.3 Halting problem

Halting problem can be found in many references [5][4][7][11][13], it can be described as: Given an arbitrary Turing machine $\mathcal{M}$, will it eventually halt when started on an arbitrary input w? We use predicate to formalize the halting problem. It in default takes a pair $(\mathcal{M}, \omega)$ ($\mathcal{M}$ represents a Turing machine, ω represents an input to that machine) as input, so the predicate must be a binary function. As the predicates we have already defined are all unary functions, we apply pairing function (see Section 2.4) to encode the input pair $(\mathcal{M}, \omega)$ and then the halting problem can be represented by an unary predicate.

Definition 5 *Predicate $\mathscr{H}$ for the halting problem*

$$\mathscr{H}(x) = \begin{cases} 1, & \text{Turing machine } \mathcal{M} \text{ encoded by } r(x) \text{ eventually halts on the input } l(x); \\ 0, & \text{otherwise.} \end{cases}$$

Theorem 3 *$\mathscr{H}$ is non-computable.*

Proof The detailed proof can be found in [14].

3.4 Halting empty tape problem

While the halting problem asks the halting behavior of arbitrary Turing machine on arbitrary input, halting empty tape problem focus on the halting behavior of arbitrary Turing machine on the empty tape input. We describe the *halting empty tape problem* as: Given an arbitrary Turing machine $\mathcal{M}$, will it eventually halt when started on an empty input?

Definition 6 *Predicate $\mathscr{E}$ for the halting empty tape problem*

$$\mathscr{E}(x) = \begin{cases} 1, & \textit{Turing machine } \mathcal{M} \textit{ encoded by } x \textit{ eventually halts on the empty input;} \\ 0, & \textit{otherwise.} \end{cases}$$

4 Equivalence of These Problems

This section is to prove the correctness of following theorem:

Theorem 4 *Predicates $\mathscr{O}$, $\mathscr{H}$, $\mathscr{E}$, and $\mathscr{S}$ are all non-computable.*

We accomplish the task by proving that these predicates are equivalent to each other under Turing computability, i.e., if there exists a Turing machine $\mathcal{M}$ that can solve any one problem of them, the remaining are easy to get a solution by constructing a new Turing machine which calls $\mathcal{M}$ as a subroutine. As we have shown that $\mathscr{H}$ is non-computable (Theorem 3), all of them are non-computable.

4.1 Reducing busy beaver problem to halting problem

Theorem 5 $\mathscr{O} \preceq_t \mathscr{H}$.

Proof Our proving approach goes like this: with $\mathscr{H}$, we generate the set $\omega(n)$; with set $\omega(n)$, we calculate $\Sigma(n)$; with $\Sigma(n)$, we solve the problem $\mathscr{O}$. Assume that there is a Turing machine $\mathcal{M}_{\mathscr{H}}$ that solves the problem $\mathscr{H}$.

Step 1. We make use of $\mathcal{M}_{\mathscr{H}}$ to construct a program $\mathcal{P}_\omega$ that calculates the function $\omega(x)$, as shown in Program 1.

Descrption of Program 1: $\mathcal{P}_\omega$ for calculating $\omega(n)$

```
   Input: a positive integer n
   Output: the set of all n-state Turing machines that halt when started on a blank tape
1  set = ∅;
2  foreach Turing machine M ∈ Ω(n) do
3      if M_H(< 0, ρ(M) >) then /* If machine M halts on empty tape */
4          add M to set;
5  return set;
```

We sketchily give an upper bound to program $\mathcal{P}_\omega$:

$$\omega \quad \in \quad O(\Omega + (4n+4)^{2n}\mathscr{H}) \tag{1}$$

where $(4n+4)^{2n}$ is the size of $\Omega(n)$. Now that $\mathscr{H}$ is computable and in Section 2.4 we said that $\Omega(x)$ can be computed in polynomial time, so from Equation 1 we can readily know that $\omega(x)$ is computable also.

Step 2. With program $\mathcal{P}_\omega$, we build program $\mathcal{P}_\Sigma$ (as shown in Program 2) that calculates the max number of 1s a n-state Turing machine can possible write on the tape before halting when

started on a empty tape, i.e., we can pick out the busy beavers from all n-state Turing machines.

Descrption of Program 2: $\mathcal{P}_\Sigma$ for calculating $\Sigma(n)$

Input: a positive integer n
Output: the max number of *1*s that a n-state Turing machine can write on the tape before halting when started on an empty tape

```
1  maxOnes = 0;
2  tmp = 0;
3  call program P_ω(n) to calculate the set ω(n);
4  foreach Turing machine M ∈ ω(n) do
5      tmp = CountOnes(ρ(M));
6      if maxOnes < tmp then /* If M writes more 1s, set it as busy beaver  */
7          maxOnes = tmp;
8  return maxOnes;
```

There are at least two x-state busy beavers (one and its reverse) in $\omega(n)$, so we can get $\Sigma(n)$ from the return value *maxOnes*. The upper bound of program $\mathcal{P}_\Sigma$ is

$$\Sigma \quad \in \quad O(\omega + |\omega(n)| \cdot (CountOnes + \rho)) \quad \in \quad O(\omega + (4n+4)^{2n} \cdot (CountOnes + \rho)) \qquad (2)$$

where $|\omega(n)|$ is the size of set $\omega(n)$, $(4n+4)^{2n}$ is the size of set $\Omega(n)$.

Step 3. We construct a program $\mathcal{P}_{\mathscr{O}}$ comparing the number of *1*s a given n-state Turing machine writes on the tape when started on a empty tape to $\Sigma(n)$, and determines whether it is a busy beaver or not. $\mathcal{P}_{\mathscr{O}}$ is shown in Program 3.

Descrption of Program 3: $\mathcal{P}_{\mathscr{O}}$ for the busy beaver problem

Input: a positive integer x which encodes a Turing machine $\mathcal{M}$
Output: if $\mathcal{M}$ is a *busy beaver* among all $\delta(x)$-state Turing machines, then output *1*; otherwise output *0*

```
1  maxOnes = 0;
   /* call P_Σ to calculate the max 1s of δ(x)-state machines that halts  */
2  maxOnes = Σ(δ(x));
3  if CountOnes(x) == maxOnes then
4      return 1;
5  else
6      return 0;
```

We have shown $\Sigma(\delta(x))$ is computable in **Step 2**, it's easy to find out that $\mathscr{O}$ is computable also.

By applying following program calling chains $\mathcal{P}_{\mathscr{H}} \Longrightarrow \mathcal{P}_\omega \Longrightarrow \mathcal{P}_\Sigma \Longrightarrow \mathcal{P}_{\mathscr{O}}$, we have proved that if there is a machine $\mathcal{M}_{\mathscr{H}}$ solving the problem $\mathscr{H}$, then we can construct a Turing machine $\mathcal{M}_{\mathscr{O}}$ involving $\mathcal{M}_{\mathscr{H}}$ to solve the problem $\mathscr{O}$.

4.2 Reducing halting problem to halting empty tape problem

Theorem 6 $\mathscr{H} \preceq_t \mathscr{E}$.

Proof Our proving approach goes like this: taking in a variable x which is an encoding of input pair ($\mathcal{M}$, ω) and transform it to a integer $f(x)$ of a single program such that machine $r(x)$ halts

on tape $l(x)$ if and only if machine $f(x)$ halts on empty tape[11]. In this way, we have $\mathscr{H}(x)$ if and only if $\mathscr{E}(f(x))$. Assume that there is a Turing machine $\mathcal{M}_{\mathscr{E}}$ that can solve the problem $\mathscr{E}$.

Step 1. We create a program $\mathcal{P}_C$ (illustrated in Program 4) which generates a new Turing machine to simulate the behavior of given Turing machine and given input[13]. Denoting the function computed by $\mathcal{P}_C$ as $C(x)$ (which means *copy*).

Descrption of Program 4: $\mathcal{P}_C$ for simulating a machine on given input

Input: a positive integer x which satisfies: 1) $r(x)$ represents a Turing machine $\mathcal{M}_i$; 2) $l(x)$ represents an input ω to $\mathcal{M}_i$.

Output: a positive integer y represents a Turing machines $\mathcal{M}_o$ accomplishes certain tasks. $\mathcal{M}_o$ works in the following strange way: 1) $\mathcal{M}_o$ starts on an empty tape, write input ω on its tape; 2) $\mathcal{M}_o$ puts itself in the running configuration of (q_s, ω), where q_s is the start state of $\mathcal{M}_i$; 3) After that, $\mathcal{M}_o$ acts just like $\mathcal{M}_i$, i.e., $\mathcal{M}_o$ simulates $\mathcal{M}_i$.

As you can see, we omit the details of program $\mathcal{P}_C$ because it's cockamamie and nonsignificant, readers who are interested can find the details in [13]. The point is that $\mathcal{P}_C$ finishes its job in polynomial time, thus function $C(x)$ is computable.

Step 2. With copy function $C(x)$, we are construct $\mathcal{P}_{\mathscr{H}}$ solving problem $\mathscr{H}(x)$.

Descrption of Program 5: $\mathcal{P}_{\mathscr{H}}$ for the halting problem

Input: a positive integer x encoding $(\mathcal{M}, \omega)$

Output: if the Turing machine represented by $r(x)$ eventually halts on the input $l(x)$ then output *1*; otherwise output *0*

```
1  copy = 0;
2  copy = C(x);
   /* If the new copy machine halts, it indicates M halts on input ω      */
3  if E(copy) then
4  |  return 1;
5  else
6  |  return 0;
```

We know from **Step 1** that $C(x)$ is computable and by assumption $\mathscr{E}(x)$ is computable, so it is trivial that $\mathscr{H}(x)$ is also computable.

4.3 Reducing halting empty tape problem to max shifter problem

Theorem 7 $\mathscr{E} \preceq_t \mathscr{S}$.

Proof Our proving approach goes like this: with predicate $\mathscr{S}$, we calculate the value *max shifts* $S(n)$; with $S(n)$, we solve the problem $\mathscr{E}$[12]. Assume that there is a Turing machine $\mathcal{M}_{\mathscr{S}}$ that can solve the problem $\mathscr{S}$.

Step 1. With program $\mathcal{P}_{\mathscr{S}}$, we are able to build a new program $\mathcal{P}_S$ (shown in Program 6) that calculates the max number of shifts a n-state Turing machine can possible take before halting when started on a empty tape, i.e., we can pick out the max shifter from all n-state Turing machines.

Descrption of Program 6: $\mathcal{P}_S$ for calculating $S(n)$

Input: a positive integer n
Output: the max number of shifts that a n-state Turing machine can take before halting when started on an empty tape

```
1  maxShifts = 0;
2  foreach Turing machine M in Ω(n) do
3      if 𝒮(ρ(M)) then /* find one max shifter, record the shifts it takes */
4          maxShifts = CountShifts(M);
5          break;
6  return maxShifts;
```

There are at least two n-state *max shifters* (one and its reverse) in $\Omega(n)$, so we can get $S(n)$ from the return value *maxShifts*. The upper bound of $\mathcal{P}_S$ is

$$S \quad \in \quad O(\Omega + |\Omega(n)| \cdot \mathscr{S} + CountShifts) \quad \in \quad O(\Omega + (4n+4)^{2n} \cdot \mathscr{S} + CountShifts) \quad (3)$$

where $|\Omega(n)|$ is the size of set $\Omega(n)$. By assumption, we have $\mathscr{S}$ is computable, so from Equation 3 $S(n)$ is computable.

Step 2. We construct a program $\mathcal{P}_{\mathscr{E}}$ (depicted in Program 7) to simulate a Turing machine $\mathcal{M}$ on empty tape by running it at most $S(\delta(\mathcal{M}))$ steps. $S(\delta(\mathcal{M}))$ is calculated by $\mathcal{P}_S$.

Descrption of Program 7: $\mathcal{P}_{\mathscr{E}}$ for the halting empty tape problem

Input: a positive integer x which encodes a Turing machine $\mathcal{M}$
Output: if $\mathcal{M}$ eventually halts on an empty tape, then output *1*; otherwise output *0*

```
1  i = 0;
   /* call P_S to calculate the max shifters of δ(x)-state machines that halts */
2  maxShifts = S(δ(x));
3  while i < maxShifts do
4      TakeOneStep(x);
5      if GetCurrentState(x) == q_halt then /* halts after taking the ith step */
6          break;
7      i = i + 1;
8  if i == maxShifts then
9      return 0;
10 else
11     return 1;
```

we give a simple upper bound to program $\mathcal{P}_{\mathscr{E}}$ as following

$$\exists N_0 \in \mathcal{N} \quad \text{such that} \quad \mathscr{E} \quad \in \quad O(S + N_0 \times (TakeOneStep + GetCurrentState)) \qquad (4)$$

where N_0 satisfies $N_0 > maxShifts$ and is a constant. We have shown that S is computable in **Step 1**, and both $TakeOneStep$ and $GetCurrentState$ are computable (defined in Section 2.4), so from Equation 4 we find that $\mathscr{E}$ is computable.

By applying following program calling chains $\mathcal{P}_{\mathscr{S}} \Longrightarrow \mathcal{P}_S \Longrightarrow \mathcal{P}_{\mathscr{E}}$, we have triumphantly proved that if there is a Turing machine $\mathcal{M}_{\mathscr{S}}$ solving the problem $\mathscr{S}$, then we can construct a Turing machine $\mathcal{M}_{\mathscr{E}}$ involving $\mathcal{M}_{\mathscr{S}}$ to solve the problem $\mathscr{E}$. That is, we have reduced $\mathscr{S}$ to $\mathscr{E}$.

4.4 Reducing max shifter problem to busy beaver problem

Theorem 8 $\mathscr{S} \preceq_t \mathscr{O}$.

Proof Our proving approach goes like this: with predicate $\mathscr{O}$, we calculate the value $\Sigma(n)$; with Σ, we get an upper bound $\mathcal{U}$ for $S(n)$; with $\mathcal{U}$, we calculates the value $S(n)$; with $S(n)$, we finally solve the problem $\mathscr{S}$. Assume that there is a Turing machine $\mathcal{M}_{\mathscr{O}}$ that solves the problem $\mathscr{O}$.

Step 1. We make use of $\mathcal{P}_{\mathscr{O}}$ to construct a program $\mathcal{P}_\Sigma$ that calculates the function $\Sigma(n)$, as described in Program 8.

Descrption of Program 8: $\mathcal{P}_\Sigma$ for calculating $\Sigma(n)$

Input: a positive integer n
Output: the max number of *1*s that a n-state Turing machine can write on the tape before halting when started on an empty tape

```
1 maxOnes = 0;
2 foreach Turing machine M in Ω(n) do
3     if O(ρ(M)) then /* If machine M is a busy beaver, record the max ones */
4         maxOnes = CountOnes(M);
5         break;
6 return maxOnes;
```

There are at least two n-state busy beavers (one and its reverse) among all n-state Turing machines, thus we can get $\Sigma(n)$ from *maxOnes*. we give an upper bound to program $\mathcal{P}_\Sigma$ briefly

$$\Sigma \quad \in \quad O(\Omega + |\Omega(n)| \cdot \mathscr{O} + CountOnes) \quad \in \quad O(\Omega + (4n+4)^{2n} \cdot \mathscr{O} + CountOnes) \qquad (5)$$

By assumption, we have $\mathscr{O}$ is computable, so from Equation 5 it is easy figure out $\Sigma(n)$ is computable.

Step 2. Rado[1] observed the inequality that the reader could easily prove

$$S(n) \leq (n+1) \cdot \Sigma(5n) \cdot 2^{\Sigma(5n)}. \qquad (6)$$

This inequality gives rise to a curious observation: *there is an upper bound $\mathcal{U}$ for $S(n)$ that is related with $\Sigma(n)$*. Better bounds have been raised and proved in detail[15][16]. With program $\mathcal{P}_\Sigma$, we construct a program to calculate the upper bound $\mathcal{U}$ for $S(x)$ based on $\Sigma(n)$. Let's mark this program as $\mathcal{P}_\mathcal{U}$, which is shown in Program 9.

Descrption of Program 9: $\mathcal{P}_\mathcal{U}$ for calculating the upper bound of max shifts

Input: a positive integer n
Output: an upper bound for the max number of shifts a n-state Turing machine can take before halting when started on an empty tape

```
1 upperBound = 0;
2 maxOnes = 0;
3 tmp = 0;
4 maxOnes = Σ(5n);
5 tmp = power(2, maxOnes);
6 upperBound = (n + 1) · maxOnes · tmp;
7 return upperBound;
```

$power(m, n)$ is a predefined function calculating m^n, obviously it is computable. Given $\Sigma(n)$ is computable from **Step 1**, $\mathcal{U}(n)$ is computable.

Step 3. Given the upper bound $\mathcal{U}(n)$ for function $S(n)$, we build program $\mathcal{P}_S$ (depicted in Program 10) that calculates the max number of shifts a n-state Turing machine can possible take before halting when started on a blank tape.

Descrption of Program 10: $\mathcal{P}_S$ for calculating $S(n)$

```
Input: a positive integer n
Output: the max number of shifts that a n-state Turing machine can take before halting
        when started on an empty tape
1  maxShifts = 0;
2  i = 0;
3  tmp = 0;
4  upperBound = U(n);
5  foreach Turing machine M in Ω(n) do
6      i = 0;
7      tmp = 0;
8      while i < upperBound do
9          TakeOneStep(M);
10         tmp = tmp + 1;
           /* Only if M can halt before taking upperBound steps, will it be a
              possible max shifter; If M does not halt after upperBound steps,
              it will never halt.                                              */
11         if GetCurrentState(M) == q_halt then
12             if maxShifts < tmp then /* find a possible max shifter          */
13                 maxShifts = tmp;
14             break;
15         i = i + 1;
16 return maxShifts;
```

As $\mathcal{U}(n)$ is an upper bound for $S(n)$, if a Turing machine $\mathcal{M}$ of n states can not halt after running $\mathcal{U}(n)$ steps, it will never halt. In program $\mathcal{P}_S$, we simulate all x-state Turing machines in set $\Omega(n)$ by running them at most $\mathcal{U}(n)$ steps. Some of these machines will halt and the other will not. *Max Shifters* are hidden in those halt, we are able to get $S(n)$.

Step 4. We construct a program $\mathcal{P}_{\mathscr{S}}$ (shown in Program 11) comparing the number of shifts a given n-state Turing machine takes before halting when started on a empty tape to $S(n)$ calculated by program $\mathcal{P}_S$. We have shown $S(x)$ is computable in **Step 3**, so $\mathscr{S}$ is computable by analysing program $\mathcal{P}_{\mathscr{S}}$.

By applying following program calling chains $\mathcal{P}_{\mathscr{O}} \Longrightarrow \mathcal{P}_{\Sigma} \Longrightarrow \mathcal{P}_{\mathcal{U}} \Longrightarrow \mathcal{P}_S \Longrightarrow \mathcal{P}_{\mathscr{S}}$, we have successfully proved that if there is a Turing machine $\mathcal{M}_{\mathscr{O}}$ solving the problem $\mathscr{O}$, then we can construct a Turing machine $\mathcal{M}_{\mathscr{S}}$ involving $\mathcal{M}_{\mathscr{O}}$ to solve the problem $\mathscr{S}$. In a word, we have reduced $\mathscr{S}$ to $\mathscr{O}$.

Descrption of Program 11: $\mathcal{P}_{\mathscr{S}}$ for the max-shifter problem

Input: a positive integer x which represents a Turing machine $\mathcal{M}$
Output: if $\mathcal{M}$ is a max shifter among all $\delta(x)$-state Turing machines, then output *1*; otherwise output *0*

```
/* call P_S to calculate the max shifts of δ(x)-state machines that halts   */
maxShiftes = S(δ(x));
if CountShifts(x) == maxShiftes then
    return 1;
else
    return 0;
```

5 Summary

We have defined four non-computable decision problems *busy beaver problem*, *max shifter problem*, *halting empty tape problem*, and *halting problem* related to the busy beaver problem defined by Rado in 1962[1] and have proven that they are equivalent to each other in terms of Turing reduction and Turing computability. The proofs of these theorems illustrate a variety of constructive techniques which are powerful in computation theory.

These theorems suggest further research into non-computable functions and the busy beaver problem. Upper bound of $S(n)$ is directly used without being proved in this paper. It's interesting to give more precise upper bounds of $\Sigma(n), S(n)$. As they are known to grow faster than any computable function, an upper bound will surely be generated from another non-computable function, this indicates a creative way to find new non-computable functions. After studying some non-computable functions and analysing their relationships with the halting problem under Turing reduction, it seems to me that there are only three cases between a given non-computable function $\mathcal{Q}$ and the halting problem $\mathcal{H}$: $\mathcal{Q} \preceq_t \mathcal{H}$, $\mathcal{H} \preceq_t \mathcal{Q}$ or $\mathcal{Q} \equiv_t \mathcal{H}$. Obviously, this is a suspectable hypothesis. Let $\mathscr{U}$ be the set of all Turing non-computable functions, then $\mathcal{H} \in \mathscr{U}$. An interesting question is that if there exists a problem $\mathcal{Q} \in \mathscr{U}$ satisfies following conjecture $\mathcal{H} \not\preceq_t \mathcal{Q}$ and $\mathcal{Q} \not\preceq_t \mathcal{H}$. My work will force on the answers to these two questions.

References

[1] T. Rado. On non-computable functions. *Bell System Tech. J*, 41(3):877–884, 1962.

[2] Alan M Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London mathematical society*, 42(2):230–265, 1936.

[3] Thomas A Sudkamp and Alan Cotterman. *Languages and machines: an introduction to the theory of computer science*, volume 2. Addison-Wesley Reading, Mass., 1988.

[4] Charles Petzold. *The annotated Turing: a guided tour through Alan Turing's historic paper on computability and the Turing machine*. Wiley Publishing, 2008.

[5] Fangmin Song. *Introduction to the Models of Computation*. China Higher Education Press, CHEP, 2012.

[6] David Barker-Plummer. Turing machines. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Summer 2013 edition, 2013.

[7] Martin Davis, Ron Sigal, and Elaine J Weyuker. *Computability, complexity, and languages: fundamentals of theoretical computer science*. Academic Press, 1994.

[8] Nigel Cutland. *Computability: An introduction to recursive function theory*. Cambridge university press, 1980.

[9] P. Bürgisser. *Completeness and reduction in algebraic complexity theory*, volume 7. Springer, 2000.

[10] M. Sipser. *Introduction to the Theory of Computation*, volume 27. Thomson Course Technology Boston, MA, 2006.

[11] John E Hopcroft. *Introduction to Automata Theory, Languages, and Computation, 3/E*. Pearson Education India, 2008.

[12] R. Machlin and Q.F. Stout. The complex behavior of simple machines. *Physica D: Nonlinear Phenomena*, 42(1):85–98, 1990.

[13] Christos H Papadimitriou. *Computational complexity*. John Wiley and Sons Ltd., 2003.

[14] M.A. Nielsen, I. Chuang, and L.K. Grover. Quantum computation and quantum information. *American Journal of Physics*, 70:558, 2002.

[15] Amir M. Ben-Amram, Bryant A. Julstrom, and Uri Zwick. A note on busy beavers and other creatures. *Mathematical systems theory*, 29(4):375–386, 1996.

[16] Amir M Ben-Amram and Holger Petersen. Improved bounds for functions related to busy beavers. *Theory of Computing Systems*, 35(1):1–11, 2002.