CS 5800 Notes: Undecidability

based on Sudkamp [3], Hopcroft, Motwani & Ullmann [1] (HMU), and Rich [2]

1 Decision problems

Previously we introduced a decision (yes/no) problem referenced as the *perfect square problem*: to determine whether or not a number $j \in \mathbb{N}$ is a perfect square ($j = k^2$ for some $k \in \mathbb{N}$). A decision problem consists of a set of questions called *instances*, each of which has a yes or no answer. The perfect square problem P_{sq} has the instances:
p_0: Is 0 a perfect square?
p_1: Is 1 a perfect square?
p_2: Is 2 a perfect square?
...
p_j: Is j a perfect square?
...

A solution to a decision problem is an algorithm that determines the answer to every instance. If such a solution exists, the problem is called *decidable*. If no such solution exists, the problem is *undecidable*.

In the chapter on Turing machines we solved the perfect square problem using a 3-tape TM that determines the answer for each j represented as an input string a^j. Doing so, the TM thus solves the associated *membership problem* in the language $L_{sq} = \{a^j \mid j \text{ is a perfect square}\}$: Is $a^j \in L_{sq}$ for some $j \in \mathbb{N}$, or, is $u \in L_{sq}$ for $u \in \Sigma^* = \{a\}^*$?

This further shows that L_{sq} is recursive as it is decided by a TM of the form depicted on the right of Fig. 1. Thus $L_{sq} \in D$, where D is the set of all recursive languages.

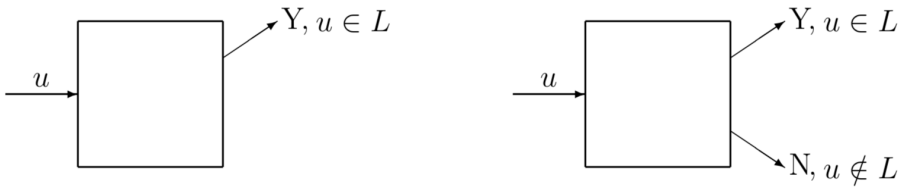


Figure 1: Left: TM for r.e. language L; Right: TM for recursive language L

The TM on the left of Fig. 1 recognizes problem instances for which the answer is "Yes" (Y). A problem for which such a TM exists is called *(Turing) semi-decidable*.

2 The Church-Turing thesis

We mentioned the Church-Turing thesis in the introduction given to Turing machines, as it justifies the TM model for computability.

Church-Turing thesis for decision problems. There is an effective procedure (algorithm) to solve a decision problem if and only if there is a Turing machine that halts for all input strings and solves the problem (decides every problem instance).

Church-Turing thesis for recognition problems. A decision problem is semi-decidable (partially solvable) if and only if there is a Turing machine that accepts precisely the problem instances for which the answer is "Yes."

Through the Church-Turing thesis, the term "Turing semi-decidabe" becomes synonymous with "semi-decidable," "Turing decidable" with "decidable," and "Turing computable" with "computable" (see Section 3).

3 Turing computable functions

A Turing computable function of one (string) argument (i.e., a unary function) is defined in [3] (Ch. 9):

Definition 3.1. *A one-tape DTM (deterministic TM)* $M = (Q, \Sigma, \Gamma, \delta, q_0, \{q_f\})$ **computes a unary function** $f : \Sigma^* \to \Sigma^*$ *if*
– *the only transition from the* q_0 *just skips to the right over the first blank on the tape;*
– *there are no transitions into* q_0;
– *there are no transitions of the form* $\delta(q_f, B)$;
– *when* $f(u) = v$, *the computation halts in the configuration* $q_f BvB$;
– *when* $f(u)$ *is undefined, denoted as* $f(u) \uparrow$, *the computation runs forever (does not halt).*
A function is **Turing computable** *or* **partial recursive** *if there is a TM that computes it.*

This TM may not halt for all inputs, since a partial recursive function may not be total (everywhere defined).

Definition 3.2. *A partial recursive function that is also total is a* **recursive function**. *In that case there is a TM that computes it and halts for all inputs.*

In Section 1 we mentioned the membership problem, i.e., for a string $u \in \Sigma^*$, determine if $u \in L$. A function associated with this problem, that returns 1 if $u \in L$, and 0 otherwise is the characterstic function of L.

Definition 3.3. *The* characterstic function *of a language* L *is a function* $\chi_L : \Sigma^* \to \{0, 1\}$ *defined as:*

$$\chi_L(u) = \begin{cases} 1 & \text{if } u \in L \\ 0 & \text{if } u \notin L \end{cases}$$

If there is a TM that computes the characterstic function of the language L, then L is recursive.

Similarly, a language L is r.e. if there is a TM that computes the *partial characterstic function* $\hat{\chi}_L$:

Definition 3.4. *The* partial characterstic function *of a language L is a function $\hat{\chi}_L : \Sigma^* \to \{0, 1\}$ defined as:*

$$\hat{\chi}_L(u) = \begin{cases} 1 & \text{if } u \in L \\ \uparrow & \text{if } u \notin L \end{cases}$$

4 Universal Turing machine

The Universal Turing machine U takes as input a representation $R(M)$ of a TM M and a string w, and simulates M on input w. It is assumed that M is a standard TM that accepts by halting, i.e., M accepts string $w \in \Sigma^*$ if its computation with input w halts (normally). The computation for any input not in $L(M)$ will not terminate.

As shown by Fig. 2, if M halts and accepts w, then U accepts $R(M)w$; if M does not halt, then U will never halt and give an answer, thus U loops.

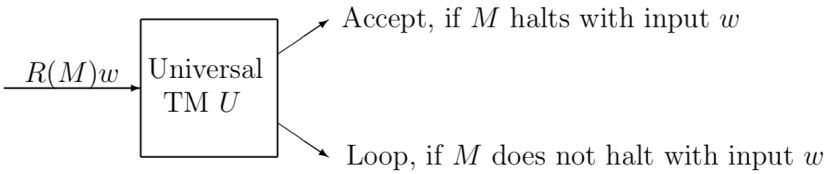


Figure 2: Universal TM U

A possible encoding to represent each transition of M as part of $R(M)$ on the tape of U is given in Table 4. Without loss of generality it can be assumed that $\Sigma = \{0, 1\}$ and $\Gamma = \{0, 1, B\}$, since other alphabet symbols can be encoded using zeros and ones.

A transition $\delta(q_i, x) = [q_j, y, d]$ is encoded as a string $en(q_i)0en(x)0en(q_j)0en(y)0en(d)$, where each $en()$ consists of a string of *1*s, and consecutive $en()$s are separated by a *0*. The transitions are separated by *00*, and the beginning and end of the sequence of transitions are marked by *000*.

An example TM M from [3] is given in Fig. 3. For instance, the transition $\delta(q_0, B) = [q_1, B, R]$ is encoded as the string *101110110111011*. A representation for M is then given by *000101110110111011001101101110110110011010101010011101101011010000*.

The language of the universal TM U, consisting of the strings $R(M)w$ accepted by U, is the *halting language* denoted by L_H or H and defined by

Definition 4.1. *The halting language is $L_H = H = \{R(M)w \mid \text{TM } M \text{ halts on input } w\}$.*

Table 1: Encoding $en()$ [3]

Symbol s	$en(s)$
0	*1*
1	*11*
B	*111*
q_0	*1*
q_1	*11*
...	...
q_i	1^{i+1}
...	...
L	*1*
R	*11*

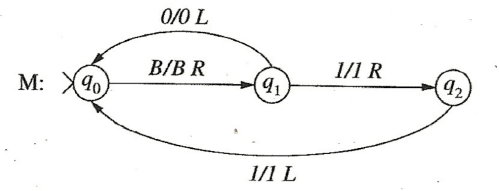


Figure 3: Example TM for encoding

Theorem 4.1. *The halting language L_H is r.e.*

Proof: TM U accepts strings $R(M)w$ of the language L_H where M is a TM that halts on string w; whereas U does not halt if M does not halt on w. Thus the TM U depicted in Fig. 2 adheres to the form on the left of Fig. 1. A construction of U as a 3-tape TM is given explicitly in [3]. It follows that L_H is r.e.

■

5 Undecidable problems

In this section we want to introduce the halting problem for TMs. However, we will start with a seemingly more practical problem that cannot be solved by a computer. This as well as several related problems address what a computer program will do when presented with an input.

5.1 The *hello world* problem

This section is based on the introduction to undecidable problems in [1]. The decision problem introduced is the *hello world* problem stated as:

Definition 5.1. *The* **hello world problem:** *Will a given (C) program P, on given input I, print "hello, world" as its first 12 characters printed?*

Susequently we will use the shorthand saying a program prints "hello, world" to mean that it prints "hello, world" as its first 12 characters printed.

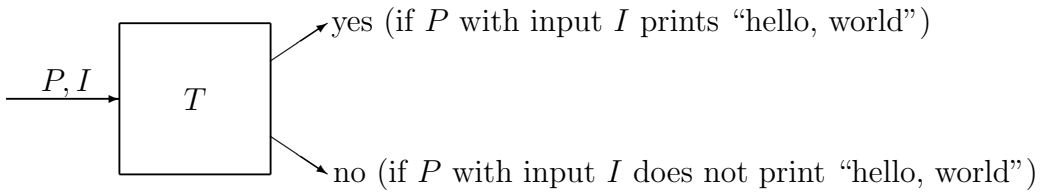
For some programs P (and input I) it is clear that P will print "hello, world" as in the classic introductory C program from Kernighan & Ritchie:

```
main() {
  printf("hello, world"\n);
}
```

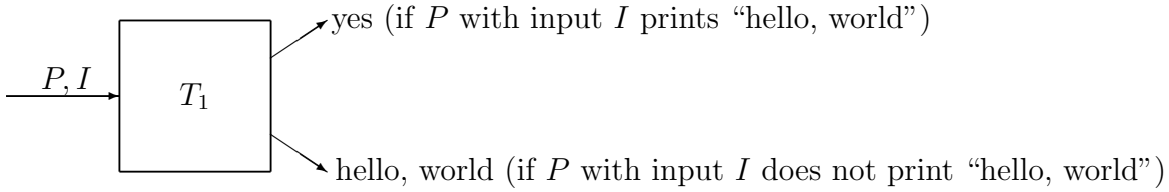
However, in other cases, the program might perform a complicated computation leading to checking a condition, and printing "hello, world" if the condition is satisfied. For such a program P it may not be known when or if the condition will ever be satisfied; thus the printing may occur after a very long time or may never happen. It will be shown that a "hello world" tester algorithm that takes as input a C program P and input I, and decides whether or not P prints "hello world" on input I, does not exist. Although it may be possible to solve the problem for particular P and I, the difficulty lies in considering arbitrary P and I.

Theorem 5.1. *The hello world problem is undecidable.*

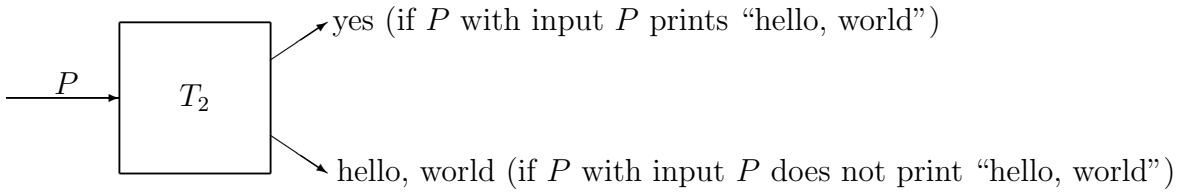
Proof: We sketch a proof from the HMU text [1]; for further details see the book. The proof is by contradiction. We start by assuming that a hypothetical *hello world* tester T exists, which decides the problem according to the figure below.



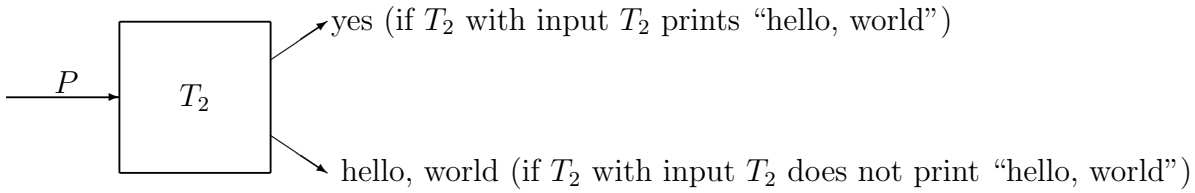
Then we perform a number of transformations to come to a contradiction. First, T is transformed into T_1, which performs exactly as T does, but prints "hello, world" exactly when T prints "no".



Then T_2 is constructed, which takes only P as input, stores a copy of P in an array, and then executes T_1 with input P, P.



Finally, a self-reference is invoked where T_2 takes itself as input. As seen the figure below, the outputs of T_2 on itself as input are paradoxal. Thus T_2 cannot exist; therefore, going back, this disproves the existence of T, showing that the problem is undecidable.



■

5.2 Reductions

A problem P_1 that is known to be undecidable may be used to prove the undecidability of another problem, P_2, if a reduction R from P_1 to P_2 can be performed, which is a Turing computable function that converts an instance of P_1 to an instance of P_2 while preserving

the yes or no answer of P_1. So the instance of P_2 is anwered yes iff the instance of P_1 is answered yes.

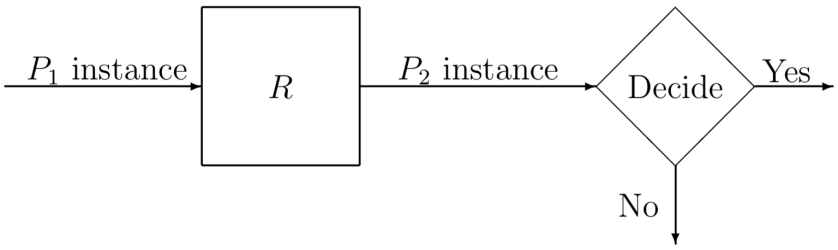


Figure 4: Proving undecidability of P_2 using reduction R from undecidable problem P_1 [1]

An argument to prove that P_2 is undecidable can then be given by contradiction. Assume that P_2 is decidable; then there is an algorithm to decide it, represented by the diamond box in Fig. 4. However, the composition of R and the P_2 solver will decide P_1, which is a contradiction since P_1 is undecidable.

As an example (from [1]), let the *hello world problem* be P_1 with as instance a (C) program P and input I. Consider the decision problem P_2 (the *call foo problem*): will a given (C) program Q with input J ever call the function foo()? The P_2 instance is constructed as follows, starting from the program P of the *hello worldi problem* instance:
– If program P already calls a function foo(), delete it and all calls to it in P; the resulting program is Q_1.
– Add a function foo() to Q_1, that does nothing and is currently not called; the resulting program is Q_2.
– Modify Q_2 to remember the first 12 characters it prints, storing them in an array A; the resulting program is Q_3.
– Modify Q_3 so that whenever it excutes an output statement, it checks the contents of the array A. If it has written "hello, world" as the first 12 characters, it calls the new function foo(); the resulting program is Q and its input J is the same as I.

Using the argument of Fig. 4 it follows that the *call foo problem* is undecidable.

Many problems about programs are undecidable, e.g., the *initializatiion problem* [2]: Can a compiler check to make sure every program variable is initialized before it is used?
The following undecidable problems are listed in [2]:
– Given a program P and input I, will P halt on input I?
– Given a program P, input I, and variable x, will P with input I ever assign a value to x?
– Given a program P, will P get into an infinite loop on some input?
– Given a program P and input I, will P with input I ever output a 0? Or anything at all?
– Given a program P and code segment S, will P ever reach S on any input, or is S dead code (i.e., can we omit S from the program)?

– Given a program P and code segment S, will P reach S on every input?
– Given two programs, are they equivalent?

The first problem in the list above is a version of the *halting problem*, which is addressed for TMs in the next section. The second is related to the *initialization problem.*

5.3 The halting problem for Turing machines

Definition 5.2. *The* **halting problem for Turing machines** *is the problem to determine, for an arbitrary TM M over an alphabet Σ and an arbitrary string w, whether M will halt on input w.*

Theorem 5.2. *The halting problem for TMs is undecidable.*

For the proof see [3]. It is given along the same lines as that of the undecidability of the *hello world problem.*

The language associated with the halting problem is the halting language denoted L_H (or H) (see Definition 4.1).

Theorem 5.3. *The halting language L_H is r.e. but not recursive, i.e. $L_H \in SD - D$.*

Proof: L_H is r.e. as given by Theorem 4.1. The non-recursiveness of L_H follows immediately from the undecidability of the halting problem.

■

As an immediate consequence, the set (D) of the recursive languages is a proper subset of the set (SD) of the recursively enumerable languages.

Theorem 5.4. *The complement of a recursive language is recursive.*

Proof: Let L be a recursive language. Thus it is accepted by a deciding TM M of the form on the right of Fig. 1. A deciding TM $\overline{M}$ for $\overline{L}$ is obtained by reversing the answers of M. Thus $\overline{M}$ answers Y = yes if $u \notin L$ (i.e., $u \in \overline{L}$), and N = no if $u \in L$ (i.e., $u \notin \overline{L}$).
The existence of $\overline{M}$ proves the decidability of $\overline{L}$.

■

Theorem 5.5. *If both a language L and its complement $\overline{L}$ are r.e., then both L and $\overline{L}$ are recursive.*

Proof: Since L and $\overline{L}$ are r.e., they are each recognized by a TM of the form on the left of Fig. TM-dec-semidec, say M_1 for L and M_2 for $\overline{L}$.
– To prove that L is recursive, a deciding TM M for L can be constructed by combining M_1 and M_2 (in parallel). The diagram of M is left as an exercise. On input u for M, if $u \in L$, we have that M_1 says "yes" and we return "yes" as the output of M; if $u \notin L$, we have that M_2 says "yes" and we return "no" as the output of M.
– Recursiveness of $\overline{L}$ can be shown similarly; also follows from Theorem 5.4.

■

Theorem 5.6. *$\overline{L_H}$ is not r.e.*

Proof: By contradiction. Assume $\overline{L_H}$ is r.e. We know L_H is r.e. By Theorem 5.5, L_H (and $\overline{L_H}$) would be recursive. But this is a contradiction with the fact that L_H is not recursive (Theorem 5.3).

■

5.4 The Post correspondence problem

The Post correspondence problem is an undecidable problem that can be described using n dominoes. The i^{th} domino has two strings, u_i and v_i, from an alphabet Σ, with u_i in the top half of the domino and v_i in the bottom half.

Definition 5.3. *Formally, a* **Post Correspondence System (PCS)** *consists of an alphabet Σ and a finite set of ordered pairs $[u_i, v_i]$ where $u_i, v_i \in \Sigma^+$ for $i = 1, 2, \ldots, n$.*

A **solution** *to a PCS is a sequence $i_1, i_2, \ldots, i_k$ such that $u_{i_1} u_{i_2} \ldots u_{i_k} = v_{i_1} v_{i_2} \ldots v_{i_k}$.*

The problem of determining whether a PCS has a solution is the **Post correspondence problem (PCP)**.

Thus, in a PCP solution, any of the dominoes can be repeated any number of times, so that the string read from left to right across the tops of the domino sequence equals the string across the bottom.

Example 5.1. Consider the PCS with $n = 4$ and the dominoes given in Fig. 5 by $[u_1, v_1] = [a, ac]$, $[u_2, v_2] = [c, ba]$, $[u_3, v_3] = [ba, a]$, $[u_4, v_4] = [acb, b]$. A solution of this PCS is given by the sequence of indexes $i_1, i_2, \ldots, i_k = 1, 2, 3, 1, 4$, which leads to $u = v = acbaaacb$ (across top and bottom), cf.i, Fig. 6. In this case, a solution can be constructed very easily. It

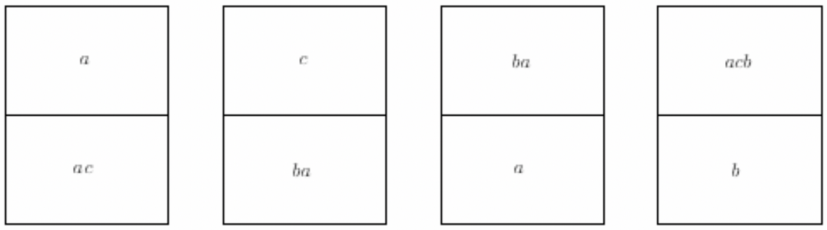


Figure 5: Sample PCS [3]

has to start with a domino that has top and bottom string beginning with the same symbol; here we have only domino 1 where u_1 and v_1 both start with a. Then we see that the top of the next domino must start with c (domino 2). Then the top of the next domino must start with ba (domino 3). The top of the next domino must start with a (domino 1). The top of the next domino must start with ac (domino 4), which completes the solution.

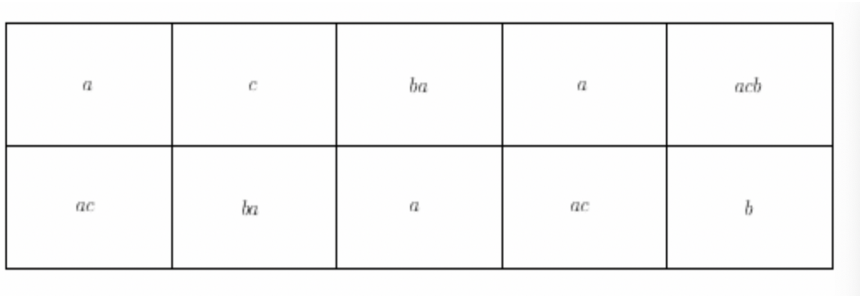


Figure 6: Solution for PCS of Fig. 5 [3]

See the text [3] for more examples. For some problems it is possible to show that the PCS has no solution, by trying out all possibilities and concluding that none lead to a solution.

The undecidability of the PCP allows to show that various problems about CFGs are undecidable (see [3] Section 12.7):
– Given an abitrary CFG G it is undecidable to determine if G is ambiguous.
– Given an abitrary CFG G it is undecidable to determine if its language $L(G) = \Sigma^*$.
– Given two arbitrary CFGs G_1 and G_2 it is undecidable to determine if their languages have any strings in common ($L(G_1) \cap L(G_2) = \emptyset$).
– Given two arbitrary CFGs G_1 and G_2 it is undecidable to determine if they derive the same language ($L(G_1) = L(G_2)$).

References

[1] Hopcroft, J. E., Motwani, R., and Ullman, J. D. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, 2nd edition, 2001. ISBN 0-201-44124-1.

[2] Rich, E. *Automata, Computability, and Complexity: Theory and Applications*. Pearson Prentice Hall, 2008. ISBN: 0-13-228806-0; ISBN: 978-0-13-228806-4.

[3] Sudkamp, T. A. *An Introduction to the Theory of Computer Science – Languages and Machines*. Pearson, Addison Wesley, 3rd edition, 2006. ISBN 0-321-32221-5.