UNIT 2 ALGORITHMICALLY UNSOLVABLE PROBLEMS

Structure **Page Nos.**

2.0 INTRODUCTION

In this unit, we discuss issues and problems that exhibit the limitations of computing devices in solving problems. *We also prove the undecidability of the halting problem.* It is related to Gödel's Incompleteness Theorem which states that there is no system of logic strong enough to prove all true sentences of number theory.

In addition, we will discuss a number of other problems, which though can be formulated properly, yet are not solvable through any computational means. And we will *prove* that such problems cannot be solved no matter what language is used, what machine is used, and how much computational resources are devoted in attempting to solve the problem etc.

2.1 OBJECTIVES

After going through this unit, you should be able to:

- show that Halting Problem is uncomputable/unsolvable/undecidable;
- to explain the general technique of *Reduction* to establish other problems as uncomputable;
- establish unsolvability of many unsolvable problems using the technique of reduction;
- enumerate large number of unsolvable problems, including those about Turing Machines and about various types of grammars/languages including context-free, context-sensitive and unrestricted etc.

2.2 DECIDABLE AND UNDECIDABLE PROBLEMS

A function g with domain D is said to be computable if there exists some Turing machine

$M = (Q, \Sigma, T, \delta, q_0, F)$ such that
$q_0\, w \overset{*}{\vdash} q_f\, g(w)$, $q_f \in F$, for all $w \in D$.

where,

$q_0\ \omega$ denotes the initial configuration with left-most symbol of the string ω being scanned in state q_0 **and** $q_f\ g(\omega)$ denotes the final c.

A function is said to be uncomputable if no such machine exists. There may be a Turing machine that can compute f on part of its domain, but we call the function computable only if there is a Turing machine that computes the function on the whole of its domain.

For some problems, we are interested in simpler solution in terms of "yes" or "no". For example, we consider problem of context free grammar i.e., for a context free grammar G, Is the language L(G) ambiguous. For some G the answer will be "yes", for others it will be "no", but clearly we must have one or the other. The problem is to decide whether the statement is true for any G we are given. The domain for this problem is the set of all context free grammars. We say that a problem is decidable if there exists a Turing machine that gives the correct answer for every statement in the domain of the problem.

Similarly, consider the problem of equivalence of context free grammar i.e., to determine whether two context free grammars are equivalent. Again, given context free grammars G_1 and G_2, the answer may be "yes" or "no". The problem is to decide whether the statement is true for any two given context free grammars G_1 and G_2. The domain for this problem is the set of all context free grammars. We say that a problem is decidable if there exists a Turing machine that gives the correct answer for every statement in the domain of the problem.

A class of problems with two outputs "yes" or "no" is said to be decidable (solvable) if there exists some definite algorithm which always terminates (halts) with one of two outputs "yes" or "no". Otherwise, the class of problems is said to be undecidable (unsolvable).

2.3 THE HALTING PROBLEM

There are many problem which are not computable. But, we start with a problem which is important and that at the same time gives us a platform for developing later results. One such problem is the halting problem. Algorithms may contain loops that may be infinite or finite in length. The amount of work done in an algorithm usually depends on the data input. Algorithms may consist of various numbers of loops, nested or in sequence. Informally, the **Halting problem** can be put as:

Given a Turing machine M and an input *w* to the machine M, determine if the machine M will eventually halt when it is given input *w*.

Trial solution: Just run the machine M with the given input *w*.

- If the machine M halts, we know the machine halts.
- But if the machine doesn't halt in a reasonable amount of time, we cannot conclude that it won't halt. May be we didn't wait long enough.

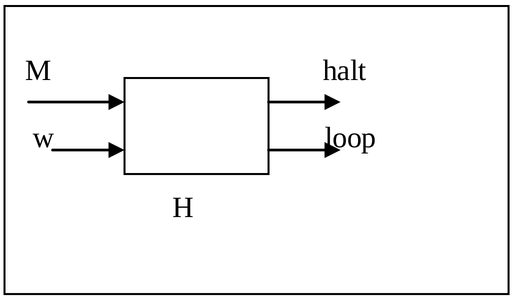
What we need is an algorithm that can determine the correct answer for any M and *w* by performing some analysis on the machine's description and the input. But, we will show that no such algorithm exists.

Let us see first, proof devised by Alan Turing (1936) that halting problem is unsolvable.

Suppose you have a solution to the halting problem in terms of a machine, say, H. H takes two inputs:

1. a program M and
2. an input *w* for the program M.

H generates an output "***halt***" if H determines that M stops on input *w* or it outputs "***loop***" otherwise.

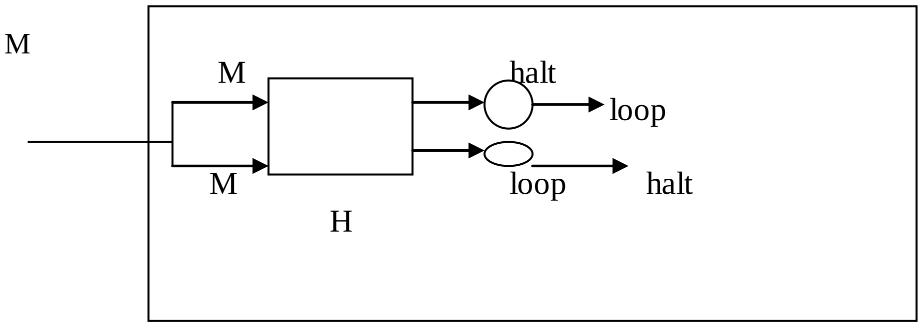


So now H can be revised to take M as both inputs (the program and its input) and H should be able to determine if M will halt on M as its input.

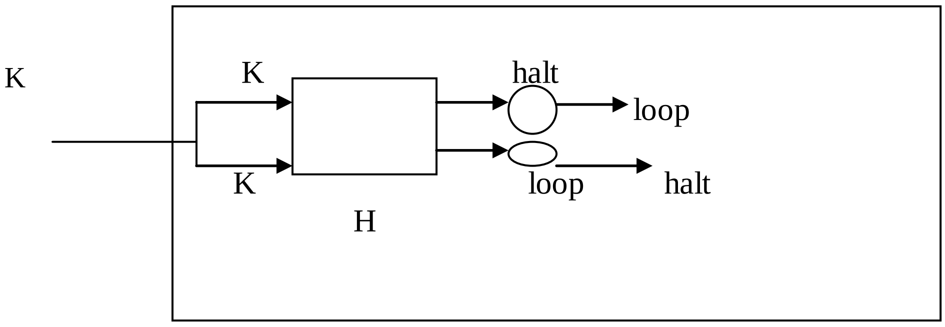
Let us construct a new, simple algorithm K that takes H's output as its input and does the following:

1. if H outputs "***loop***" then K halts,
2. otherwise H's output of "***halt***" causes K to loop forever.

That is, K will do the **opposite** of H's output.



Since K is a program, let us use K as the input to K.



If H says that K halts then K itself would loop (that's how we constructed it).
If H says that K loops then K will halt.

In either case H gives the wrong answer for K. **Thus H cannot work in all cases.**

We've shown that it is possible to construct an input that causes any solution H to fail. Hence, The halting problem is undecidable.

Now, we formally define what we mean by the halting problem.

Definition 1.1: Let W_M be a string that describes a Turing machine M = (Q, Σ, T, δ, q_0, F), and let *w* be a string in Σ^*. We will assume that W_M and w are encoded as a string of 0's and 1's. A solution of the halting problem is a Turing machine H, which for any W_M and *w*, performs the computation

$q_0 W_M w \vdash^* x_1 q_y x_2$ if M applied to w halts, and

$q_0 W_M w \vdash^* y_1 q_n y_2$ if M applied to w does not halt.

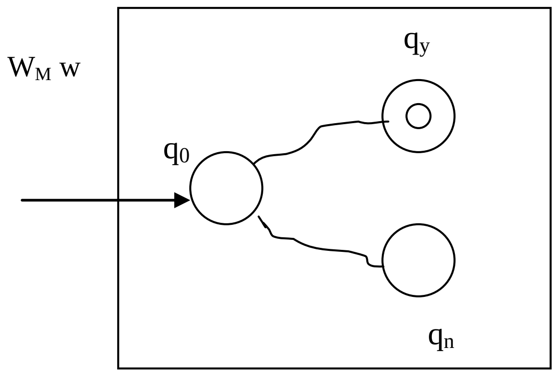
Here q_y and q_n are both final states of H.

Theorem 1.1: There does not exist any Turing machine H that behaves as required by Definition 1.1. The halting problem is therefore undecidable.

Proof: We provide proof by contradiction. Let us assume that there exists an algorithm, and consequently some Turing machine H, that solves the halting problem. The input to H will be the string W_M *w*. The requirement is then that, the Turing machine H will halt with either a yes or no answer. We capture this by asking that H will halt in one of two corresponding final states, say, q_y or q_n. We want H to operate according to the following rules:

$q_0 W_M w \vdash^*_M x_1 q_y x_2$ if M applied to *w* halts, and
$q_0 W_M w \vdash^*_M y_1 q_n y_2$ if M applied to *w* does not halt.

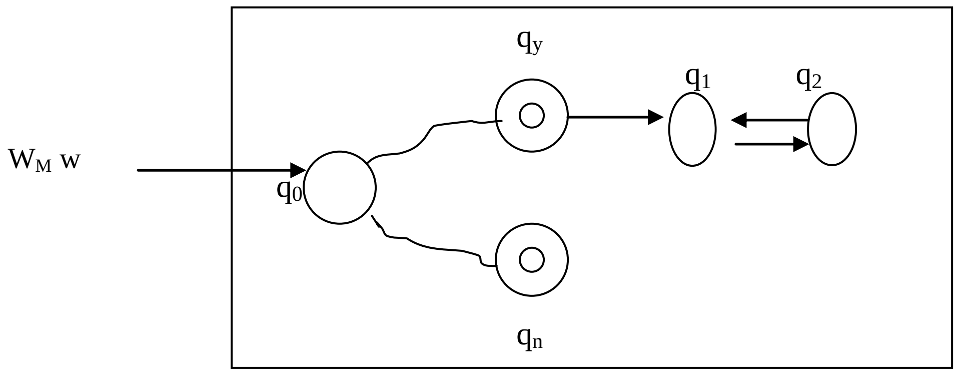
This situation can also be visualized by a block diagram given below:



Next, we modify H to produce H_1 such that

- If H says that it will halt then H_1 itself would loop
- If H says that H will not halt then H_1 will halt.

We can achieve this by adding two more states say, q_1 and q_2. Transitions are defined from q_y to q_1, from q_1 to q_2 and from q_2 to q_1, regardless of the tape symbol, in such a way that the tape remains unchanged. This is shown by another block diagram given below.



Formally, the action of H_1 is described by

q_0 W_M w $\vdash^*_{H1}$ ∞ if M applied to w halts, and

q_0 W_M w $\vdash^*_{H1}$ $y_1 q_n y_2$ if M applied to w does not halt.

Here, ∞ stands for Turing machine is in infinite loop *i.e.*, Turing machine will run forever. Next, we construct another Turing machine H_2 from H_1. This new machine takes as input W_M and copies it, ending in its initial state q_0. After that, it behaves exactly like H_1. The action of H_2 is such that

q_0 W_M $\vdash^*_{H2}$ q_0 W_M W_M $\vdash^*_{H2}$ ∞ if M applied to W_M halts, and

q_0 W_M $\vdash^*_{H2}$ $y_1 q_n y_2$ if H_2 applied to W_M does not halt.

This clearly contradicts what we assumed. In either case H_2 gives the wrong answer for W_M. Thus H cannot work in all cases.

We've shown that it is possible to construct an input that causes any solution H to fail. Hence, the halting problem is undecidable.

Theorem 2.2: If the halting problem were decidable, then every recursively enumerable language would be recursive. Consequently, the halting problem is undecidable.

Proof: Recall that

1. A language is *recursively enumerable* if there exists a Turing machine that accepts every string in the language and does not accept any string not in the language.

2. A language is *recursive* if there exists a Turing machine that accepts every string in the language and rejects every string not in the language.

Let L be a recursively enumerable language on Σ, and let M be a Turing machine that accepts L. Let us assume H be the Turing machine that solves the halting problem. We construct from this following algorithm:

1. Apply H to W_M w. If H says "no", then by definition w is not in L.

2. If H says "yes", then apply M to w. But M must halt, so it will ultimately tell us whether w is in L or not.

This constitutes a membership algorithm, making L recursive. But, we know that there are recursively enumerable languages that are not recursive. The contradiction implies that H cannot exist *i.e.,* the halting problem is undecidable.

2.4 REDUCTION TO ANOTHER UNDECIDABLE PROBLEM

Once we have shown that the halting problem is undecidable, we can show that a large class of other problems about the input/output behaviour of programs are undecidable.

Examples of undecidable problems

- **About Turing machines**:
  - Is the language accepted by a TM empty, finite, regular, or context-free?
  - Does a TM meet its "specification ?" that is, does it have any "bugs."
- **About Context Free languages**
  - Are two context-free grammars equivalent?
  - Is a context-free grammar ambiguous?

Not so surprising, Although this result is sweeping in scope, may be it is not too surprising. If a simple question such as whether a program halts or not is undecidable, why should one expect that any other property of the input/output behaviour of programs is decidable? Rice's theorem makes it clear that failure to decide halting implies failure to decide any other interesting question about the input/output behaviour of programs. Before we consider Rice's theorem, we need to understand the concept of problem reduction on which its proof is based.

Reducing problem B to problem A means finding a way to convert problem B to problem A, so that a solution to problem A can be used to solve problem B.

One may ask, Why is this important? A reduction of problem B to problem A shows that problem A is at least as difficult to solve as problem B.Also, we can show the following:

- To show that a problem A is undecidable, we reduce another problem that is known to be undecidable to A.
- Having proved that the halting problem is undecidable, we use problem reduction to show that other problems are undecidable.

Example 1: Totality Problem

Decide whether an arbitrary TM halts on all inputs. (If it does, it computes a "total function"). This is equivalent to the problem of whether a program can ever enter an infinite loop, for any input. It differs from the halting problem, which asks whether it enters an infinite loop for a particular input.

Proof: We prove that the halting problem is reducible to the totality problem. That is, if an algorithm can solve the totality problem, it can be used to solve the halting problem. Since no algorithm can solve the halting problem, the totality problem must also be undecidable.
The reduction is as follows. For any TM M and input *w*, we create another TM M_1 that takes an arbitrary input, ignores it, and runs M on *w*. Note that M_1 halts on all inputs if and only if M halts on input *w*. Therefore, an algorithm that tells us whether

M_1 halts on all inputs also tells us whether M halts on input *w*, which would be a solution to the halting problem.

Hence, The totality problem is undecidable.

Example 2: Equivalence problem

Decide whether two TMs accept the same language. This is equivalent to the problem of whether two programs compute the same output for every input.

Proof: We prove that the totality problem is reducible to the equivalence problem. That is, if an algorithm can solve the equivalence problem, it can be used to solve the totality problem. Since no algorithm can solve the totality problem, the equivalence problem must also be unsolvable.

The reduction is as follows. For any TM M, we can construct a TM M_1 that takes any input *w*, runs M on that input, and outputs "yes" if M halts on *w*. We can also construct a TM M_2 that takes any input and simply outputs "yes." If an algorithm can tell us whether M_1 and M_2 are equivalent, it can also tell us whether M_1 halts on all inputs, which would be a solution to the totality problem.

Hence, the equivalence problem is undecidable.

Practical implications

- The fact that the totality problem is undecidable means that we cannot write a program that can find any infinite loop in any program.
- The fact that the equivalence problem is undecidable means that the code optimization phase of a compiler may improve a program, but can never guarantee finding the optimally efficient version of the program. There may be potentially improved versions of the program that it cannot even be sure are equivalent.

  We now describe a more general way of showing that a problem is undecidable i.e., **Rice's theorem**. First we introduce some definitions.
- A *property* of a program (TM) can be viewed as the set of programs that have that property.
- A *functional (or non-trivial) property* of a program (TM) is one that some programs have and some don't.

Rice's theorem (proof is not required)

- "Any functional property of programs is undecidable."
- A functional property is:
  (i) a property of the input/output behaviour of the program, that is, it describes the mathematical function the program computes.
  (ii) nontrivial, in the sense that it is a property of some programs but not all programs.

Examples of functional properties

- The language accepted by a TM contains at least two strings.
- The language accepted by a TM is empty (contains no strings).

- The language accepted by a TM contains two different strings of the same length.

Rice's theorem can be used to show that whether the language accepted by a Turing machine is context-free, regular, or even finite, are undecidable problems.
Not all properties of programs are functional.

2.5 UNDECIDABILITY OF POST CORRESPONDENCE PROBLEM

Undecidable problems arise in language theory also. It is required to develop techniques for proving particular problems undecidable. In 1946, Emil Post proved that the following problem is undecidable:

Let Σ be an alphabet, and let L and M be two lists of nonempty strings over Σ, such that L and M have the same number of strings. We can represent L and M as follows:

$$L = (w_1, w_2, w_3, ..., w_k)$$
$$M = (v_1, v_2, v_3, ..., v_k)$$

Does there exist a sequence of one or more integers, which we represent as (i, j, k, ..., m), that meet the following requirements:

- Each of the integers is greater than or equal to one.
- Each of the integers is less than or equal to k. (Recall that each list has k strings).
- The concatenation of $w_i, w_j, w_k, ..., w_m$ is equal to the concatenation of $v_i, v_j, v_k, ..., v_m$.

If there exists the sequence (i, j, k, ..., m) satisfying above conditions then (i, j, k, ..., m) is a solution of PCP.

Let us consider some examples.

Example 3: Consider the following instance of the PCP:

Alphabet Σ = { a, b }
List L = (a, ab)
List M = (aa, b)

We see that (1, 2) is a sequence of integers that solves this PCP instance, since the concatenation of a and ab is equal to the concatenation of aa and b (i.e ,$w_1 w_2 = v_1 v_2$ = aab). Other solutions include: (1, 2, 1, 2) , (1, 2, 1, 2, 1, 2) and so on.

Example 4: Consider the following instance of the PCP Alphabet Σ = { 0, 1 }

List L = (0, 01000, 01)
List M = (000, 01, 1)

A sequence of integers that solves this problem is (2, 1, 1, 3), since the concatenation of 01000, 0, 0 and 01 is equal to the concatenation of 01, 000, 000 and 1 (i.e., $w_2 w_1 w_1 w_3 = v_2 v_1 v_1 v_3$ =010000001).

2.6 UNDECIDABLE PROBLEMS FOR CONTEXT FREE LANGUAGES

The Post correspondence problem is a convenient tool to study undecidable questions for context free languages. We illustrate this with an example.

Theorem 1.2: There exists no algorithm for deciding whether any given context-free grammar is ambiguous.

Proof : Consider two sequences of strings $A = (u_1, u_2, \dots , u_m)$ and $B = (v_1, v_2, \dots , v_m)$ over some alphabet Σ. Choose a new set of distinct symbols a_1, $a_2, \dots , a_m$, such that

$$\{a_1, a_2, \dots , a_m\} \cap \Sigma = \varnothing,$$

and consider the two languages

$L_A = \{ u_iu_j, \dots\ u_l\, u_k\, a_k\, a_l \dots , a_j\, a_i\}$ defined over A and $\{a_1, a_2, \dots , a_m\}$

and

$L_B = \{ v_iv_j, \dots\ v_l\, v_k\, a_k\, a_l \dots , a_j\, a_i\}$ defined over B and $\{a_1, a_2, \dots , a_m\}$.

Let G be the context free grammar given by

$$(\{S, S_A , S_B\}, \{a_1, a_2, \dots , a_m\} \cup \Sigma,\ P, S)$$

where the set of productions P is the union of the two subsets: the first set P_A consists of

$S \rightarrow S_A$,
$S_A \rightarrow u_iS_Aa_i \mid u_ia_i$, $\quad i = 1, 2,\dots, n$,

the second set P_B consists of

$S \rightarrow S_B$,
$S_B \rightarrow v_iS_Ba_i \mid v_ia_i$, $\quad i = 1, 2,\dots, n$.

Now take

$G_A = (\{S, S_A \}, \{a_1, a_2, \dots , a_m\} \cup \Sigma,\ P_A, S)$

and

$G_B = (\{S, S_B \}, \{a_1, a_2, \dots , a_m\} \cup \Sigma,\ P_B, S)$

Then,
$L_A = L(G_A)$,
$L_B = L(G_B)$,
and
$L (G) = L_A \cup L_B$.

It is easy to see that G_A and G_B by themselves are unambiguous. If a given string in L (G) ends with a_i, then its derivation with grammar G_A must have started with $S \Rightarrow u_i S_A a_i$. Similarly, we can tell at any later stage which rule has to be applied. Thus, If G is ambiguous it must be because there is w for which there are two derivations

$$S \Rightarrow S_A \Rightarrow u_i S a_i \Rightarrow^* u_i u_j \ldots u_k a_k \ldots a_j a_i = w$$

and

$$S \Rightarrow S_B \Rightarrow v_i S a_i \Rightarrow^* v_i v_j \ldots v_k a_k \ldots a_j a_i = w.$$

Consequently, if G is ambiguous, then the Post correspondence problem with the pair (A, B) has a solution. Conversely, If G is unambiguous, then the Post correspondence problem cannot have solution.

If there existed an algorithm for solving the ambiguity problem, we could adapt it to solve the Post correspondence problem. But, since there is no algorithm for the Post correspondence problem, we conclude that the ambiguity problem is undecidable.

2.7 OTHER UNDECIDABLE PROBLEMS

- Does a given Turing machine M halt on all inputs?
- Does Turing machine M halt for *any* input? (That is, is L(M)=∅?)
- Do two Turing machines M_1 and M_2 accept the same language?
- Is the language L(M) finite?
- Does L(M) contain any two strings of the same length?
- Does L(M) contain a string of length k, for some given k?
- If G is a unrestricted grammar.
- Does L(G) = ∅ ?
- Does L(G) infinite ?
- If G is a context sensitive grammar.
- Does L(G) = ∅ ?
- Does L(G) infinite ?
- If L_1 and L_2 are any context free languages over Σ.
- Does $L_1 \cap L_2 = \emptyset$?
- Does $L_1 = L_2$?
- Does $L_1 \subseteq L_2$?
- If L is recursively enumerable language over Σ.
- Does L empty ?
- Does L finite ?

Ex. 1) Show that the state-entry problem is undecidable.

Hint: The problem is described as follows: Given any Turing machine M = (Q, Σ, T, δ, q_0, F) and any $q \in Q$, $w \in \Sigma^+$, to determine whether Turing machine M, when given input w, ever enters state q.

Ex. 2) Show that the blank tape halting problem is undecidable.

Hint: The problem is described as follows: Given a Turing machine M, Does Turing machine M halts when given a blank input tape?

Ex. 3) Consider the following instance of the PCP:

Alphabet $\Sigma = \{ 0, 1, 2 \}$
List L = (0, 1, 2)
List M = (00, 11, 22)
Does PCP have a solution ?

Ex. 4) Consider the following instance of the PCP:

Alphabet $\Sigma = \{ a, b \}$
List L = (ba, abb, bab)
List M = (bab, bb, abb)
Does PCP have a solution ?

Ex. 5) Does PCP with two lists A = (b, babbb, ba) and B = (bbb, ba, a) have a solution ?

Ex. 6) Does PCP with two lists A = (ab, b, b) and (abb, ba, bb) have a solution ?

Ex.7) Show that there does not exist algorithm for deciding whether or not

$L(G_A) \cap L(G_B) = \varnothing$ for arbitrary context free grammars G_A and G_B.

2.8 SUMMARY

- A decision problem is a problem that requires a yes or no answer. A decision problem that admits no algorithmic solution is said to be undecidable.
- No undecidable problem can ever be solved by a computer or computer program of any kind. In particular, there is no Turing machine to solve an undecidable problem.
- We have not said that undecidable means we don’t know of a solution today but might find one tomorrow. It means we can never find an algorithm for the problem.
- We can show no solution can exist for a problem A if we can reduce it into another problem B and problem B is undecidable.

2.9 SOLUTIONS/ANSWERS

Ex. 1)

The problem is described as follows: Given any Turing machine M = (Q, Σ, T, δ, q_0, F) and any $q \in Q$, $w \in \Sigma^+$, to determine whether Turing machine M, when given input w, ever enters state q.

The problem is to determine whether Turing machine M, when given input w, ever enters state q.

The only way a Turing machine M halts is if it enters a state q for which some transition function $\delta(q_i, a_i)$ is undefined. Add a new final state Z to the Turing machine, and add all these missing transitions to lead to state Z. Now use the (assumed) state-entry procedure to test whether state Z is ever entered when M is given input w. This will reveal whether the original machine M halts. We conclude that it must not be possible to build the assumed state-entry procedure.

Ex. 2)

It is another problem which is undecidable. The problem is described as follows: Given a Turing machine M, does Turing machine M halts when given a blank input tape?

Here, we will reduce the blank tape halting problem to the halting problem. Given M and w, we first construct from M a new machine M_w that starts with a blank tape, writes w on it, then positions itself in configuration q_0w. After that, M_w acts exactly like M. Hence, M_w will halt on a blank tape if and only if M halts on w.

Suppose that the blank tape halting problem were decidable. Given any M and w, we first construct M_w, then apply the blank tape halting problem algorithm to it. The conclusion tells us whether M applied to w will halt. Since this can be done for any M and w, an algorithm for the blank tape halting problem can be converted into an algorithm for the halting problem. Since the halting problem is undecidable, the same must be true for the blank tape halting problem.

Ex. 3)

There is no solution to this problem, since for any potential solution, the concatenation of the strings from list L will contain half as many letters as the concatenation of the corresponding strings from list M.

Ex. 4)

We can not have string beginning with w_2 = abb as the counterpart v_2 = bb exists in another sequence and first character does not match. Similarly, no string can begin with w_3 = bab as the counterpart v_3 = abb exists in another sequence and first character does not match. The next choice left with us is start the string with w_1 = ba from L and the counterpart v_1 = bab from M. So, we have

ba

bab

The next choice from L must begin with b. Thus, either we choose w_1 or w_3 as their string starts with symbol b. But, the choice of w_1 will make two string look like:

baba

babbab

While the choice of w_3 direct to make choice of v_3 and the string will look like:

babab

bababb

Since the string from list M again exceeds the string from list L by the single symbol 1, a similar argument shows that we should pick up w_3 from list L and v_3 from list M. Thus, there is only one sequence of choices that generates compatible strings, and for this sequence string M is always one character longer. Thus, this instance of PCP has no solution.

Ex. 5)

We see that (2, 1, 1, 3) is a sequence of integers that solves this PCP instance, since the concatenation of babbb, b, b and ba is equal to the concatenation of ba, bbb, bbb and a (i.e., $w_2\, w_1\, w_1\, w_3 = v_2\, v_1\, v_1\, v_3$ = babbbbbba).

Ex. 6)

For each string in A and corresponding string in B, the length of string of A is less than counterpart string of B for the same sequence number. Hence, the string generated by a sequence of strings from A is shorter than the string generated by the sequence of corresponding strings of B. Therefore, the PCP has no solution.

Ex. 7)

Proof : Consider two grammars

$G_A = (\{ S_A \}, \{a_1, a_2, \dots , a_m\} \cup \Sigma,\ P_A, S_A)$

and

$G_B = (\{S_B \}, \{a_1, a_2, \dots , a_m\} \cup \Sigma,\ P_B, S_B)$.

where the set of productions P_A consists of
$S_A \rightarrow u_iS_Aa_i \mid u_ia_i$, $\quad i = 1, 2,\dots, n$,

and the set of productions P_B consists of

$S_B \rightarrow v_iS_Ba_i \mid v_ia_i$, $\quad i = 1, 2,\dots, n$.

where consider two sequences of strings $A = (u_1, u_2, \dots , u_m)$ and $B = (v_1, v_2, \dots , v_m)$ over some alphabet Σ. Choose a new set of distinct symbols $a_1, a_2, \dots , a_m$, such that

$\{a_1, a_2, \dots , a_m\} \cap \Sigma = \varnothing$,

Suppose that $L(G_A)$ and $L(G_B)$ have a common element, i.e.,

$S_A \Rightarrow u_iSa_i \Rightarrow^* u_i\, u_j \dots u_ka_k \dots a_ja_i$

and

$S_B \Rightarrow v_iSa_i \Rightarrow^* v_i\, v_j \dots v_ka_k \dots a_ja_i$.

Then the pair (A, B) has a PC-solution. Conversely, if the pair does not have a PC- solution, then $L(G_A)$ and $L(G_B)$ cannot have a common element. We conclude that $L(G_A) \cap L(G_B)$ is nonempty if and only if (A, B) has a PC- solution.

2.10 FURTHER READINGS

1. *Elements of the Theory of Computation*, H.R. Lewis & C.H.Papadimitriou: PHI, (1981).

2. *Introduction to Automata Theory, Languages, and Computation* (II Ed.), J.E. Hopcroft, R.Motwani & J.D.Ullman: Pearson Education Asia (2001).

3. *Introduction to Automata Theory, Language,* J.E. Hopcroft and J.D. Ullman: *and Computation,* Narosa Publishing House (1987).

4. *Introduction to Languages and Theory of Computation*, J.C. Martin: Tata-Mc Graw-Hill (1997).

5. *Computers and Intractability– A Guide to the Theory of NP-Completeness,* M.R. Garey & D.S. Johnson, W.H. Freeman and Company (1979).