Decidability

Contents

[Acceptance Problem for DFAs 2](#_Toc132658856)

[Acceptance Problem for NFAs 3](#_Toc132658857)

[Emptiness Problem for DFAs 3](#_Toc132658858)

[Equivalence Problem for DFAs 4](#_Toc132658859)

[Acceptance Problem for CFGs 5](#_Toc132658860)

[Emptiness Problem for CFGs 5](#_Toc132658861)

[Equivalence Problem for CFGs 7](#_Toc132658862)

[Acceptance Problem for TMs 7](#_Toc132658863)

We have seen that Turing machines can be used to solve any problem. For some language , if the Turing machine halts on all inputs of the language, the language is said to be **decidable**.

## Acceptance Problem for DFAs

When talking about decidability, one of the first questions to occur is how do we know if a language is decidable? This brings us to our first problem, called the **acceptance problem for DFAs**. The problem itself is described as follows:

To prove this, we need to create a TM that decides the problem, . The algorithm for this is:

On input ,

1. Check if has the form . If not, reject.
2. Simulate the computation of on .
3. If ends at an accept state, then accept. Otherwise reject.

For every problem going forward, we will first need to check if it has the **correct form**. Instead of writing it out every time, we can simply say ‘On input ’.

For the above problem, notice that we just proved that this language is **decidable**. A DFA takes a single step on each character of its input. will never enter an infinite loop. It will at most take a number of steps equal to the length of . At the end of , must be in either an accept or a reject state (even if that reject is implicit). Thus, is a decidable language.

## Acceptance Problem for NFAs

On input :

1. Convert into the equivalent DFA, .
2. Run TM on input .
3. Accept if accepts. Otherwise, reject.

The reason we convert the NFA to a DFA is to avoid loops. An NFA has transitions, which can result in us looping forever for some NFAs. Converting it to a DFA gets rid of those transitions, and since every NFA has an equivalent DFA, this conversion can be done.

Here, is the TM that was used to decide the problem . We are using as a **subroutine**. Since we are performing a conversion and giving it that same TM which we already showed works with a decidable language, must also be decidable.

## Emptiness Problem for DFAs

The emptiness problem requires us to check if the DFA accepts any strings at all or whether it rejects everything (i.e., ).

On input :

1. Mark the start state.
2. Repeat until no new state is marked:
   1. Mark every state that has an incoming arrow from a previously marked state.
3. Accept if none of the accept states are marked. Reject otherwise.

Here, we are simply checking if there is **some path** from the start of the DFA to an accept state. We do this by marking the states we have already visited. If, at the end of it, none of the accept states are marked, it means we were unable to find any path at all that reaches the accept state. This means the DFA **reject all inputs**, so the **problem is accepted**. Notice that in this case, we accept the problem if the DFA rejects everything, because that is what the problem asks for.

## Equivalence Problem for DFAs

On input :

1. Construct a DFA where (i.e., the set of strings where and are not the same).
2. Run on .
3. Accept if accepts. Otherwise reject.

Thus, we are checking if the set of strings that cause and to be different is an empty set.

## Acceptance Problem for CFGs

On input :

1. Convert to CNF.
2. Try all derivations of length .
3. Accept if any of the derivations are . Otherwise reject.

The reason we are checking only the derivations of length is because there is a lemma which states that if is in CNF and , then every possible derivation of has exactly steps. Since we have a bound, the problem is decidable.

There is a **corollary** that can be proven from this: “Every CFL is decidable.”

Suppose we have some CFL, , that was generated using a CFG, . On any input where , we can use to prove that the problem is decidable.

The above also in turn shows that is decidable.

## Emptiness Problem for CFGs

To prove this, we have to work backwards from the terminals. Suppose we have a simple CFG like this:

First, we can mark every terminal symbol:

Next, we mark every string that goes to a list of completely marked symbols.

Notice that will be marked next. Thus, this language is not empty. The algorithm for this is as follows:

On input :

1. Mark all terminal symbols in .
2. Repeat until there is nothing new to mark:
   1. Mark all occurrences of variable if where is marked.
3. Reject if the start variable is marked. Accept otherwise.

## Equivalence Problem for CFGs

This problem is not decidable. The reason we cannot use the same mechanism we did to decide the equivalence of DFAs is that CFLs are not closed under complement and intersection.

There are also other problems that are not decidable, such as testing if a CFG is ambiguous, i.e., a string can be generated in two possible ways.

## Acceptance Problem for TMs

This is not a decidable problem. However, it is **recognizable**. If we have some TM, , that recognizes :

On input :

1. Simulate on .
2. Accept if halts and accepts.
3. Reject if halts and rejects.

The reason we are calling this recognizable is because it is possible to simulate , but we do not know for sure whether the simulation will ever halt.