Deterministic TMs are as powerful as nondeterministic TMs

1. To show that nondeterminism does not increase the power of the Turing machine model we need to describe a general procedure by which given a nondeterministic TM

\[ N = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}}) \]

we can construct a deterministic TM

\[ D = (Q', \Sigma, \Gamma', \delta', q'_0, q'_{\text{accept}}, q'_{\text{reject}}) \]

such that

\[ L(N) = L(D) \]

and \( D \) halts on \( w \) iff all of \( N \)'s possible computations on \( w \) are finite (i.e., its computation tree is finite).

2. To do this, we will construct a deterministic machine, \( D \), that will enumerate all possible configurations that would appear in the computation tree of the nondeterministic machine \( N \) in breadth first order. Eventually, our deterministic machine’s tape will be filled with a long sequence of configurations. Configurations corresponding to nodes at the top of the computation tree will appear at the beginning of the tape. As the machine is allowed to compute longer and longer, configurations corresponding to deeper and deeper levels in the tree will be written on the end of the tape.

The algorithm to follow is simple to express if we assume \( D \) has two tapes:

- Convert input \( w \) into an initial configuration for \( N \), \((q_0, \epsilon, w)\). This will all happen on tape 1. Tape 2 will still be empty.
- Repeat (until you see an accept or reject configuration or there are no unexpanded configurations on tape 1):
  - Copy first unexpanded configuration from tape 1 to tape 2 (overwrite any previous configuration on tape 2).
  - Expand the configuration on tape 2 by writing all configurations derivable from this configuration at the end of tape 1. This will require building knowledge of \( N \)'s transition function into the transition function for \( D \).
- The tape alphabet \( \Gamma' \) that we use for \( D \) will have to contain all the symbols in \( \Gamma \) together with:
  - Some delimiter to serve in the role of the commas that separate the components of a configuration.
  - Symbols that can be used to encode all the states of \( N \).
  - Two delimiters we can use to separate configurations from one another on our tape. One will be used in the section of tape containing configurations that have been expanded ($$), the other will be used between configurations that still need to be expanded ($#$$).

3. The machine \( D \), will clearly recognize the same language as \( N \) since it will find an accepting configuration if and only if one occurs in the computation tree.

4. The machine \( D \), will clearly recognize the same language as \( N \) since it will find an accepting configuration if and only if one occurs in the computation tree.

5. If the computation tree is finite, \( D \) will eventually halt. Therefore, if \( N \) decides a language, \( D \) will decide the same language.

Languages about Automata

1. We are getting close to the goal of showing an example of a problem (i.e., language) that is not computable (i.e., decidable). The first example of such a language (and many of the examples of such languages) is a language that involves statements about automata.
2. To get ready for such languages, it makes sense to spend a little time talking about languages that make simpler statements about automata. That is, to talk about decidable languages about automata.

3. A few examples of the sorts of languages I have in mind include:
   - \( A_{DFA} = \{ \langle B, w \rangle \mid B \text{ is a DFA and } w \in L(B) \} \)
   - \( A_{REX} = \{ \langle R, w \rangle \mid R \text{ is a regular expression and } w \in L(R) \} \)
   - \( E_{DFA} = \{ \langle A \rangle \mid A \text{ is a DFA and } L(A) = \emptyset \} \)
   - \( EQ_{DFA} = \{ \langle A, B \rangle \mid A \text{ and } B \text{ are DFAs and } L(A) = L(B) \} \)
   - \( A_{CFG} = \{ \langle G, w \rangle \mid G \text{ is a CFG and } w \in L(G) \} \)
   - \( E_{CFG} = \{ \langle G \rangle \mid G \text{ is a CFG and } L(G) = \emptyset \} \)
   - \( EQ_{CFG} = \{ \langle G, H \rangle \mid G \text{ and } H \text{ are CFGs and } L(A) = L(B) \} \)

4. In the descriptions of all of these sets, the angle brackets, \( \langle \ldots \rangle \), are included to suggest that descriptions of the grammars, automata, etc. that are included in these languages must somehow be encoded in a precise way.

   - To determine that any of these languages is Turing-recognizable or Turing-decidable, we would need to describe some Turing machine that recognized the language in questions. This machine would have some fixed alphabet \( \Sigma \), but we wish to include machines and grammars over all alphabets in these languages. Accordingly, we will have to somehow encode arbitrary finite alphabets in some single alphabet.

5. The exact encoding scheme used is usually unimportant and rarely explicitly discussed in the proof of a language’s decidability, but before we start ignoring these details, I thought it would be helpful to think concretely about how we might represent one of these languages. So, let’s think about how we might represent the strings in \( A_{CFG} \).

   - This mainly boils down to how do we represent an arbitrary CFG in some fixed alphabet.

   - I would like you all to take a few minutes to design a scheme for representing arbitrary CFGs given a fixed alphabet like:

     \[
     \Sigma = \{ \rightarrow, \#, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, V, T \}
     \]

   - As an example, consider the grammar \( G = \)

     \[
     \begin{align*}
     L & \rightarrow 1L1 | + R \\
     R & \rightarrow 1R1 | =
     \end{align*}
     \]

     That is, be prepared to show how to construct the representation of \( <G, 1+1=1> \)

   - There are many ways to do this. For example, we could number the elements of the alphabets of terminals and variables (non-terminals) and then use “Tn” to encode the nth terminal and “Vn” for the nth variable.

   - We would then give up on the | symbol and just list all of the productions separated by # as in:

     \[
     V1 \rightarrow T1V1T1\#V1 \rightarrow T2V2\#V2 \rightarrow T1V2T1\#V2 \rightarrow T3
     \]

   - We could complete the representation by adding w separated by the remaining symbols with a pair of #’s.

     \[
     V1 \rightarrow T1V1T1\#V1 \rightarrow T2V2\#V2 \rightarrow T1V2T1\#V2 \rightarrow T3\#T1T2T3T1
     \]

   - Of course, we could have used binary numbers instead of decimal, or unary, or ...

6. All of the language we are interested in should only contain strings that are valid given the representation scheme we have chosen. For example, given our scheme for CFGs, a string that contained two consecutive T’s should immediately be rejected.

7. Fortunately, for reasonable representation schemes, the language of syntactically correct encodings is regular, context-free, or at worst decidable, so we can assume that any TM we describe to recognize one of these languages begins by rejecting anything with invalid format.

   **Decidable Properties of DFAs/Reg Exps.**
1. OK. Enough of that! Let’s get back to talking about language based on questions about automata, grammars, etc.

2. Assuming now that the particular representation we use is not critical, let’s consider whether some of the problems mentioned above are decidable.

3. Consider the first language:

\[ A_{DFA} = \{ \langle B, w \rangle \mid D \text{ is a DFA and } w \in \mathcal{L}(B) \}. \]

4. We can show that this language is decidable by arguing that a TM can read in a description of a DFA from its input tape and then simulate that DFA. We will assume that the representation used for a DFA is essentially a list of triples of the form \((q, x, \delta(q, x))\) describing the machine’s transition function together with descriptions of its initial state, its final states and the number of states and symbols in its alphabet.

To make the explanation of the simulation as simple as possible, we will assume a 3-tape machine.

- First, the machine will find \(w\) at the end of its first/input tape, copy it to its second tape, and erase it from the first tape, leaving mainly a list of transition function triples on the first tape.
- Next, the machine will write the description of the initial state on its third tape.
- Mainly, the machine will repeatedly (until it reaches the last input symbol):
  - Scan its first tape to find a \((q, x, \delta(q, x))\) with \(x\) matching the symbol encoded starting with the position of the second tape head and \(q\) matching the symbol on the third tape.
  - Move the second tape head to the beginning of the next encoded symbol
  - Copy the state \(\delta(q, x)\) from the triple found on the first tape over the state that had been stored on the third tape.

5. Consider the language

\[ A_{REX} = \{ \langle R, w \rangle \mid R \text{ is a regular expression and } w \in \mathcal{L}(R) \}. \]

6. Next, let’s think a bit about how we could argue the the following two languages are decidable.

- \(E_{DFA} = \{ \langle A \rangle \mid A \text{ is a DFA and } \mathcal{L}(A) = \emptyset \}, \) and
- \(A_{CFG} = \{ \langle G, w \rangle \mid G \text{ is a CFG and } w \in \mathcal{L}(G) \}. \)

7. Both of these can be shown using results we showed (much) earlier.

8. For \(E_{DFA}\), we can use the pumping lemma to argue that if \(\mathcal{L}(A) \neq \emptyset\), then there must be some \(w \in \mathcal{L}(A)\) with \(|w| \leq |Q|\) (where \(Q\) is the DFA’s set of states. We can easily construct a TM that enumerates all strings over the DFAs alphabet that are shorter than \(|Q| + 1\) in alphabetical order and uses the machine for \(A_{DFA}\) as a function/method to check if any of these strings belong to \(\mathcal{L}(A)\).

9. Alternately, we can treat \(E_{DFA}\) as a graph search problem. Viewing the machine’s state diagram as a graph, if there is any path from the start symbol to a final state in the graph, then the machine must accept the string whose symbols label the edges of the path. A breadth-first search of the graph will find all reachable states.

**Recursive, Recognizable, not even Recognizable, and worse!**

1. We have not considered one of the languages in our original list:

- \(EQ_{CFG} = \{ \langle G, H \rangle \mid G \text{ and } H \text{ are CFGs and } \mathcal{L}(A) = \mathcal{L}(B) \} \)
2. In addition, we have not considered the decidability of similar languages about TMs:

- \( A_{TM} = \{ \langle M, w \rangle \mid M \text{ is a TM and } w \in \mathcal{L}(M) \} \)
- \( E_{TM} = \{ \langle M \rangle \mid M \text{ is a TM and } \mathcal{L}(M) = \emptyset \} \)
- \( EQ_{TM} = \{ \langle M, N \rangle \mid M \text{ and } N \text{ are TMs and } \mathcal{L}(M) = \mathcal{L}(N) \} \)

3. In case you cannot guess, these are all examples of languages that are not decidable. In fact, one of them is not even Turing-recognizable.

4. Before proving these facts, I want to take some time to make sure we are as comfortable as we can get with the terminology we will be using to distinguish different degrees of computability relative to the Turing machine model.

5. We will focus on three types of languages. The diagram below summarizes the inclusion relationships that exist between these three sets of languages and several others we have (or will) encounter.

**Turing-decidable** We say a language \( L \) is Turing-decidable if there exists a TM \( M \) that halts on all inputs for which \( L = L(M) \). The terms “decidable” and “recursive” are synonymous with “Turing-decidable”. All of the languages we discussed so far this semester including regular and context-free language belong to this group.

**Turing-recognizable** We say a language \( L \) is Turing-recognizable if there exists a TM \( M \) for which \( L = L(M) \). Basically, we have simply dropped the requirement that \( M \) halt on inputs that are not in \( L \). Therefore, all Turing-decidable languages are also Turing-recognizable. The terms “recognizable” and “recursively enumerable” are synonymous with “Turing-recognizable”.

- It should be clear that \( A_{TM} = \{ \langle M, w \rangle \mid M \text{ is a TM and } w \in \mathcal{L}(M) \} \) is Turing-recognizable. Just as a TM can simulate the computation of a DFA, we can design a TM to simulate any other TM as long as it is given a description of that TM and the input that machine should process. If \( w \in L(M) \), then the simulation will eventually reach an accepting state and the simulator can accept \( \langle M, w \rangle \). However, it is not clear that this language is decidable since if \( M \) does not halt on \( w \), there is no obvious way that the simulator can distinguish this from a very long computation that will eventually accept so it will not halt either.

Note: This is not intended as a proof that \( A_{TM} \) is not decidable. The goal is just to help us clearly understand the structure of these classes of language before we do prove that various language fall into various categories.

**Not Turing-recognizable** Each Turing-recognizable language is associated with one or more (actually always more) Turing machines. We know that there are only countably many TMs. Therefore, there can only be countably many Turing-recognizable languages. Since there are uncountably many subsets of the set of all strings over any alphabet, there must be many languages that are not Turing-recognizable (a.k.a. not recursively enumerable). Intuitively, it should seem likely that

\[ E_{TM} = \{ \langle M \rangle \mid M \text{ is a TM and } \mathcal{L}(M) = \emptyset \} \]
is such a language. The only obvious way to determine that the
language of a TM is empty is to check every string to make sure
that there is no string that the machine accepts. Since there
are infinitely many strings, this process will not terminate if the
language is actually empty.

The diagram below summarizes the language we have suggested might
belong to each of these three sets of languages.