Theory of Computation Part 2: Turing Machines and Formal Languages

Michael Levet

January 14, 2014

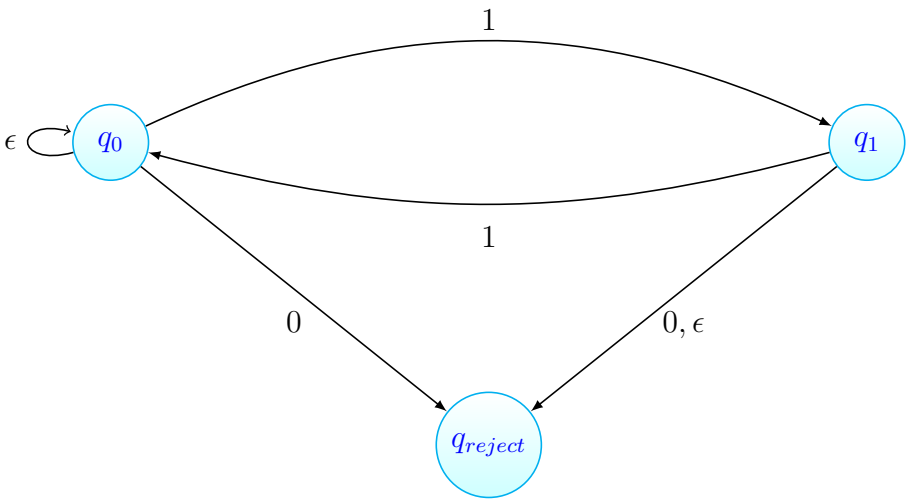
Introduction

This tutorial will expand on Part 1. In a lot of ways, this tutorial will be more practical than the last one. It will cover designing Turing Machines to accept languages, as well the concept of acceptance vs. decidability. By the conclusion of this tutorial, the reader will begin to see both the power and limitations of computing models.

Turing Machines as Language Acceptors

The first part of the tutorial will introduce Turing Machines with respect to language acceptance. That is, examples will be explored to construct Turing Machines to accept languages, such as regular and context free grammars. The purpose of this section is to develop intuition regarding the construction of Turing Machines, as well as to begin to piece together the idea that Turing Machines are capable of simulating automaton that we see in the context of Formal Languages (such as a finite state machine, pushdown automaton, etc.). In the next sections, the power and limits of Turing Machines with respect to language acceptance will be more formally and precisely discussed.

Consider an example of a regular language, over the alphabet $\{0,1\}$: $L = \{1^{2k} : k \in \mathbb{N} \cup \{0\}\} = (11)^*$. That is, there are an even number of 1's. It is easy to construct a finite state automaton to recognize this language. There are three states: q_0, q_1, and q_{reject}. Start at q_0, and transition to q_1 on an input of 1. If the string is empty, stay at q_0 and accept the string. Otherwise, transition to q_{reject}. On q_1, on the input of 1, transition to q_0. Otherwise, transition to q_{reject}. The string is accepted if and only if the finite state machine halts on q_0. The finite state machine diagram is pictured below.



Now let's construct a Turing Machine to accept $(11)^*$. The construction of the Turing Machine, is in fact, almost identical to that of the finite state automaton. The Turing Machine will start with the input string on the far-left of the tape, with the tape head at the start of the string. The Turing

Machine has $Q = \{q_0, q_1, q_{reject}, q_{accept}\}$. Let the Turing Machine start at q_0 and read in the character under the tape head. If it is not a 1 or the empty string, enter q_{reject} and halt. Otherwise, if the string is empty, enter q_{accept} and halt. On the input of a 1, transition to q_1 and move the tape head one cell to the right. While in q_1, read in the character on the tape head. If it is a 1, transition to q_0 and move the tape head one cell to the right. Otherwise, enter q_{reject} and halt. The Turing Machine always halts, and accepts the string if and only if it halts in state q_{accept}.

Notice the similarities between the Turing Machine and finite state automaton. The intuition should follow that any language accepted by a finite state automaton (ie., any regular language) can also be accepted by a Turing Machine.

Let's now modify the language, so $M = \{\omega : \omega \text{ has an even number of 1's }\}$, where the alphabet is still $\{0, 1\}$. Notice that our new language M isn't restricted to only 1's. So 00, 0110, and 1100 are all valid strings. Let's construct a Turing Machine to accept this language. The idea will be to count the 1's like in the Turing Machine to accept L. The difference will be that the Turing Machine to accept M will have to filter out the 0's to do this.

The Turing Machine to accept M will have the same states as the Turing Machine accepting L: $Q = \{q_0, q_1, q_{reject}, q_{accept}\}$. It starts with the input string left-aligned on the tape, with the tape head at the start of the string. The Turing Machine begins in state q_0, where it reads in the first character. If the string is empty, enter q_{accept} and halt. Otherwise, if there is a 0, stay at q_0 and move the tape head one cell to the right. If there is a 1, enter q_1 and move the tape head one cell to the right. While in state q_1, enter q_{reject} and halt if the input character is the empty string. If the character is a 0, stay at q_1 and move the tape head one cell to the right. Otherwise, if the input character is a 1, enter q_0 and move one cell to the right. Thus, the Turing Machine always halts and accepts M.

Consider another language: $N = \{0^n 1^n : n \in \mathbb{N} \cup \{0\}\}$. This is a context-free grammar, accepted by a pushdown automaton. The pushdown automaton would have the following states: $Q = \{q_0, q_1, q_2, q_{accept}, q_{reject}\}$. The pushdown automaton starts with an empty string and an empty stack in state q_0. If the string is empty, the pushdown automaton transitions to qaccept and halts. If the first character is not zero, the the pushdown automaton enters qreject and halts. Otherwise, the pushdown automaton pushes a 0 onto the stack and transitions into q_1. While in q_1, the pushdown automaton rejects the string if the input is empty. On an input of 0, the pushdown automaton stays in q_1 and pushes a 0 onto the stack. On an input of 1, the pushdown automaton pops a 0 from the stack and transitions to q_2. While in q_2, on an input of 1, the pushdown automaton pops a 0 from the stack. If the stack is empty or on an input of 0, the pushdown automaton enters qreject. Otherwise, it stays in q_2. The pushdown automaton enters qaccept if and only if it is in q_2, receives the empty string as input, and the stack is empty.

Now let's construct a Turing Machine to accept N. The Turing Machine has a tape alphabet of $\Gamma = \{0, 1, a, b\}$, and will function differently than the pushdown automaton. Conceptually, rather than using a stack, the Turing Machine will use its tape head. It starts with a 0, then looks for a corresponding 1. It then reads in the next 0, then looks for another 1. If it finds an unpaired 0 or 1, it rejects the string.

So initially the Turing Machine starts at q_0 with the input string on the far-left of the tape, with the tape head above the first character. If the string is empty, the Turing Machine enters qaccept and halts. If the first character is a 1, the Turing Machine enters qreject and halts. If the first character is 0, the Turing Machine replaces it with an a. It then moves the tape head to the right one cell at a time until it finds a 1. If no 1 is found, the Turing Machine rejects the string. Otherwise, the Turing Machine replaces the 1 with a b. It then moves the tape head to the left cell by cell, until it finds the first a. It then moves the tape-head to the right one cell. If there is a 0, it starts from the beginning at the current tape cell. Otherwise, it scans the string for any unmarked characters. If it finds them, it rejects the string. Otherwise, it accepts the string and halts.

Recursive vs. Recursively Enumerable, and Algorithms
This section will introduce important terminology and concepts used to discuss how Turing Machines evaluate languages. These concepts provide a foundation for understanding some of the most important issues in theoretical computer science- the power and limits of computers.

With respect to Languages, Turing Machines answer the question- is a string ω in the language L? There are three possibilities. A language L is called decidable if there exists a Turing Machine such that for all strings, the Turing Machine will correctly determine whether or not a string is in the language. That is, if $\omega \in L$, there exists a Turing Machine which will accept ω; and if $\omega \notin L$, the Turing Machine will reject ω.

The second possibility is that the language L is recursively enumerable. That is, there exists a Turing Machine such that for all strings ω, if $\omega \in L$, then the Turing Machine will accept ω. Note that if L is recursively enumerable, there is no guarantee that the corresponding Turing Machine will halt or explicitly reject strings not in L. Instead, if a string is not in L, then the Turing Machine will either explicitly reject the input or fail to halt. Such languages that have Turing Machines that halt on all inputs are decidable languages; that is, decidable languages are a subset of recursively enumerable languages.

Finally, a language L is called non-Turing Acceptable if no Turing Machine will accept all the strings in L.

The differences between decidable, recursively enumerable, and non-Turing Acceptable languages are at the foundation of what computers can and cannot do. Before proceeding, it is important to define an algorithm formally. You, the reader, have probably heard the term "algorithm" before and are familiar with what it means, conceptually. You know bubblesort is an algorithm, and you have likely implemented your own algorithms. Conceptually, it is just a finite sequence of steps that produces a specific result. This is actually quite close to the definition of an algorithm. The exact definition of an algorithm is a Turing Machine that halts on all inputs. This isn't, interesting, in and of itself though. It is quite easy to define a Turing Machine to reject all strings, or to instead accept all strings. Correctness and accomplishment is important, too. That is, algorithms should correctly do something useful.

Thinking of algorithms as Turing Machines is important, as it allows for the formalization of problems as languages. As an example, the Traveling Salesman Problem can be represented as a language: $L_{TSP} = \{G(V, E, W), \omega : \omega$ is the encoding of the optimal Hamiltonian circuit of a weighted graph $G(V, E, W)$ with a Hamiltonian Circuit $\}$. Thus, an algorithm to solve the Traveling Salesman Problem is a Turing Machine that decides L_{TSP}. Often times, a second, more precise definition of an algorithm is employed when discussing problems. A problem represented as a language, L, is decidable if there exists a Turing Machine M that decides L and $L = L(M)$. That is, the Turing Machine M accepts only the strings in L.

The set of decidable languages is closed under the following operations: set union, set intersection, set complementation, string concatenation, and Kleene closure. This means that given any two decidable languages, L_1 and L_2, applying these operators to the languages results in a set that is still decidable. Consider set union as an example. If a string is in $L_1 \cup L_2$, it is sufficient to take the Turing Machines that decide L_1 and L_2, call them M_1 and M_2 respectively, and run them on the input string. If at least one of them accepts the input string, then it is in the language $L_1 \cup L_2$. Otherwise, it isn't in the resulting language. Thus, the union of decidable languages is decidable. The logic for string concatenation, set complementation, and set intersection is quite similar.

The logic as to why the Kleene closure of a decidable language is decidable, is a bit more complicated.

Let L be a decidable language. By definition of the Kleene closure, L^* contains all possible finite strings formed from elements in L. Let M^* be a Turing Machine designed to accept L^*, and let M be the Turing Machine to decide L. M^* starts by taking an element $\omega \in L^*$ and simulating M on ω. If M reaches the accepting halt state, then that substring of ω is in L. M^* repeats this process on remainder of ω starting from one character over from the position of the character on which M halted. If M at any time rejects a string, M^* rejects ω, as a substring of ω is not in L. Otherwise, if M^* reaches the end of ω without rejecting it, M^* accepts ω. Thus, M^* accepts ω if and only if $\omega \in L^*$, and M^* rejects ω if and only if it is not in L^*. Thus, M^* decides L^*.

One final concept will be explored in this section- the concept of co-recursively enumerable. That is, let L be a recursively enumerable language, and let L' be L's complement. The languages L and L' are decidable if and only if L' is recursively enumerable. Let's conceptualize this. Since L is recursively enumerable, there exists a Turing Machine M that accepts L. Similarly, if L' is recursively enumerable, there exists a Turing Machine M' that accepts L'. So take an arbitrary string ω. By definition of set complements, $\omega \in L \oplus \omega \in L'$. Let K be a Turing Machine. K will simulate both M and M' on ω. If the simulation of M halts and accepts ω, then $\omega \in L$. Otherwise, M' will halt and accept ω. Thus, L and L' can both be decided. Since decidable languages are a subset of recursively enumerable languages, it is trivial to show that if L and L' are decidable, then L' is recursively enumerable.

Now consider if L is recursively enumerable but L'' is not recursively enumerable. Since L' is not recursively enumerable, there exists no Turing Machine to accept L'. Thus, L' is not Turing Acceptable.

Conclusion

This tutorial was designed to provide an introduction into constructing Turing Machines, as well as some introductory theory and intuition as to their limits. The next tutorial in their sequence will construct languages and further explore more concrete examples of what Turing Machines can and cannot accept or decide.