0.1 Enumerating Σ^*

The usual way to enumerate strings in Σ^* is to order them first by their length and then within strings of the same length to order them in dictionary order, as shown below.

0	λ	3	c	6	ac	...
1	a	4	aa	7	ba	
2	b	5	ab	8	bb	

Figure 1: Enumerating Σ^* with $\Sigma = \{a, b, c\}$.

A natural question that arises is what is the nth string in this enumeration? What effective procedure yields the nth string?

One way to find the nth string is to build a tree of all the strings in a "breadth-first" fashion. The first few steps are shown below.

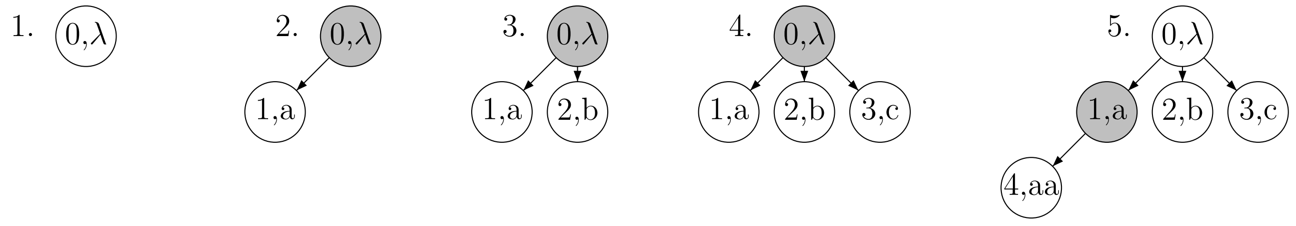


Figure 2: Enumerating Σ^* with $\Sigma = \{a, b, c\}$.

The procedure for Σ could be stated as follows. Remember we know there are k elements in Σ, and we can assume they are ordered. We are given as input a number n and we want to output the nth string in the enumeration of Σ^*.

1. Set a counter variable c to 0.
2. BUILD a node labeled $(0, \lambda)$.
3. If $0 = n$ then OUTPUT λ and STOP.
4. Otherwise, ADD $(0, \lambda)$ to the QUEUE.
5. REMOVE the first element (m, w) from the QUEUE.
6. Set variable i to 1.
7. Let a be the ith symbol in Σ.
8. Increase c by 1.
9. BUILD a node labeled $(c, w \cdot a)$ as a daughter to (m, w).
10. If $c = n$ then OUTPUT $w \cdot a$ and STOP.
11. Otherwise, ADD this daughter node to the end of the QUEUE.
12. Increase i by 1.
13. If $i > k$ then go to step 5. Otherwise, go to step 7.

The general form of this algorithm is very useful. Recall that an enumeration of Σ^* is also an enumeration of all programs! This means we could try running some set of inputs X on all the programs to find a program that gives a certain output. Basically, in steps 2 and 7 we would check to see how the program w behaves on the inputs in X. If the behavior is what we like, we output this program and stop. Otherwise we continue to the next program!

0.2 Example: Learning from positive data

We consider an example very similar to the number guessing game. Instead of a set of numbers, this time we are thinking of a set of strings. The a priori knowledge we have is that the set contains all strings except one. Is there a winning guessing rule for these stringsets? Formally, is there an algorithm the identifies this class of stringsets in the limit from positive data?

Formally, we set up the problem this way. For all $x \in \Sigma^*$, let $\overline{x} = \{w \in \Sigma^* \mid w \notin x\}$, which also equals $\Sigma^* - \{x\}$. Then we can define the class of stringsets we want to learn as $C = \{\overline{x} \mid x \in \Sigma^*\}$.

The basic idea for the learing algorithm is the same as it was for the winning guessing rule when the game involved numbers. Informally, pick the first string x in the enumeration that has not yet been observed and output a program for $\overline{x}$. Since we can enumerate the strings, we can always find the first one not yet seen.

More formally, we can adapt the algorithm above as follows. The algorithm below takes as input a finite sequence of positive examples up to time t, denoted $\varphi\langle t\rangle$, and outputs a program which solves a membership problem. Call this algorithm A-BAR (since we will show it solves the problem of learning the "bar" class C.)

1. Set a counter variable c to 0.
2. BUILD a node labeled $(0, \lambda)$.
3. **If $\varphi\langle t\rangle$ does not contain λ then OUTPUT a program solving the membership problem for $\overline{\lambda}$ and STOP.**
4. Otherwise, ADD $(0, \lambda)$ to the QUEUE.
5. REMOVE the first element (m, w) from the QUEUE.
6. Set variable i to 1.
7. Let a be the ith symbol in Σ.
8. Increase c by 1.
9. BUILD a node labeled $(c, w \cdot a)$ as a daughter to (m, w).
10. **If $\varphi\langle t\rangle$ does not contain $w \cdot a$ then OUTPUT a program solving the membership problem for $\overline{w \cdot a}$ and STOP.**
11. Otherwise, ADD this daughter node to the end of the QUEUE.
12. Increase i by 1.
13. If $i > k$ then go to step 5. Otherwise, go to step 7.

Here are few important observations.

- The only difference between outputting the nth string and this this algorithm are in the bold-faced steps 3 and 10.
- Checking whether a string x is contained in $\varphi\langle t\rangle$ is straightforward. String x is compared to each $\varphi(i)$ for all i between 1 and t inclusive. If any equal x then $\varphi\langle t\rangle$ contains x. If none equal x then it does not.
- Outputting a program which solves the membership problem for $\overline{x}$ is also straightforward. For each x, here is such a program. It takes any string y as input and outputs YES or NO as follows. If $x = y$ output NO. Otherwise, output YES.
- This is not an efficient algorithm.

Next we prove the following theorem.

Theorem 1. *A-BAR identifies C in the limit from positive data.*

Proof Consider any stringset $S \in C$ and any positive presentation φ of S. It is sufficient to show there exists some time t such that for all $m \geq t$, A-BAR($\varphi\langle m\rangle$)=A-BAR($\varphi\langle t\rangle$) and A-BAR($\varphi\langle m\rangle$) is a program solving the membership problem for S.

By definition of C, there is exactly one string x not in S and $S = \overline{x}$.

Let $E : \mathbb{N} \to \Sigma^*$ be the standard enumeration for Σ^*. It follows that there exists some $n \in \mathbb{N}$ such that $E(n) = x$. It also follows that there are only finitely many strings prior to x in the enumeration. In other words, the enumeration looks like this: $E(0), E(1), E(2), \ldots$ $E(n-1), x, E(n+1)\ldots$

Note that for each i between 0 and $n-1$ inclusive, the string $E(i)$ belongs to $\overline{x}$. Since φ is a positive presentation for $\overline{x}$, the string $E(i)$ thus occurs at some point in φ. More precisely, for each i, there is some point in time t_i such that $\varphi(t_i) = E(i)$ and for all $t < t_i$, $\varphi(t_i) \neq E(i)$. In other words, t_i is the first occurence of $E(i)$ in the positive presentation.

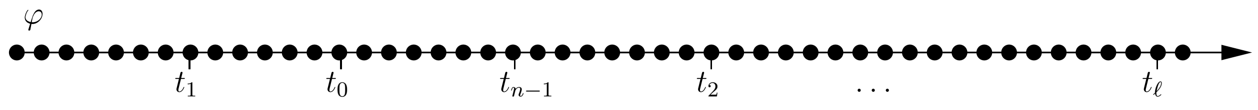
Let t_ℓ be the largest number among these t_i. Since there are finitely many t_i, this largest number exists.

We now show that for all $m \geq t_\ell$ that A-BAR($\varphi\langle m\rangle$) outputs a program that solves the membership problem for $\overline{x}$. Then we show that t_ℓ is the convergence point for A-BAR on φ.

Consider any $m \geq t_\ell$ and consider A-BAR($\varphi\langle m\rangle$). A-BAR only stops and outputs a program when it encounters a string in the standard enumeration that is not contained in $\varphi\langle m\rangle$. Since m is greater than t_ℓ, it follows that $E(0), E(1), E(2), \ldots, E(n-1)$ all occur in $\varphi\langle m\rangle$. However $E(n) = x$ does not occur in φ as it is a positive presentation of $\overline{x}$. Thus x is the first string in the enumeration that is not contained in $\varphi\langle m\rangle$. Hence A-BAR stops and outputs a program solving the membership problem for $\overline{x} = S$.

Finally, consider A-BAR($\varphi\langle t_\ell - 1\rangle$). Recall that $\ell < n$ and for all $t < t_\ell$, $\varphi(t) \neq E(\ell)$. Thus $E(\ell)$ is not contained in the $\varphi\langle t_\ell - 1\rangle$. Furthermore, since $\ell < n$ A-BAR will stop and output a program solving the membership problem for $\overline{E(\ell)}$, which is incorrect.

It follows that t_ℓ is the convergence point at and after which A-BAR always outputs a program solving the membership problem for $S = \overline{x}$. □



$E(0), E(1), E(2), \dots E(\ell), \dots E(n-1), x, \dots$

Figure 3: Illustrating the the logic of the proof. The first occurence of string $E(\ell)$ occurs at t_ℓ and is the last word prior to x in the enumeration to be observed.

There are many respects in which this learning algorithm is not like human-language learning. For example, human languages are not like any S in C. And human language-learning is not enumerative.

But there are some respects in which it is like human language-learning.

- Every stringset S belonging to C is of infinite size.
- There are infinitely many stringsets in C.
- Nonetheless, A-BAR only makes finitely many errors before it converges to a correct grammar.

This is a very simple example that illustrates how one can write a learning algorithm that provably solves a non-trivial learning problem. Language is very complicated and language-learning more complicated yet. As with many fields, it is important to understand the simple cases like the one here before tackling the more complex stuff.