Language and Computation

LING 227 01 / 627 01 / PSYC 327 01
Assignment #2
Due: February 18, 2014

The solutions to the problem set must be handed in on paper at the beginning of the class. It must be typed (printed), including mathematical formulae.

The goals of the homework are manifold: they should help you practice certain concepts, deepen your understanding of the material, test your understanding of the textbook, but also prepare you for topics and notions to be introduced later in the course. In particular, the current homework aims at: solidifying your knowledge of FSTs and edit distance, as well as practicing working with algorithms, pseudo-codes and mathematical concepts.

Each problem set is worth 10 points in total.

Problem 1: Algorithm for FST parsing

(based on JM p. 82, exercise 3.7) (4 points)

Part 1: FST as translator Let $\mathcal{T}$ be a deterministic finite-state transducer (FST), with the following properties: For all states $q \in Q$ and input letter $i \in \Sigma$, the transition and output functions $\delta(q, i)$ and $\sigma(q, i)$ are either not defined, or take single values $q' \in Q$ and $o \in (\Delta \cup \{\epsilon\})$, respectively. That is, given its current state and the next character on the input tape, $\mathcal{T}$ has no choice: it will either stop, or perform a transition to q' and write o to the output tape. Moreover, $\delta(q, \epsilon)$ and $\sigma(q, \epsilon)$ are not defined, the FST is ϵ-free on the input tape, even though it can emit ϵ (i.e., write the empty string, not write anything to the output tape).

Implement such a FST $\mathcal{T}$, by modifying the algorithm D-Recognize in Chapter 2 of your textbook (p. 5 of the January 30 lecture slides). Your algorithm will have two parameters, a tape (input string) and the FST $\mathcal{T}$, while it will return a string on the output tape (instead of an `accept` or `reject` message).

Part 2: FST as recognizer Write the algorithm for parsing a (general) finite-state transducer, using the pseudocode introduced in Chapter 2. You should do this by modifying the algorithm ND-Recognize in Fig. 2.19, Chapter 2 of your textbook (also available on pp. 8-9 of the lecture slides of January 30).

Your algorithm, fed with an FST $\mathcal{T}$, will read two strings, $a \in \Sigma^*$ and $b \in \Delta^*$. It will return `accept` or `reject`, depending on whether the pair (a, b) is in the regular relation defined by $\mathcal{T}$.

Problem 2: *Edit distance* and your intuition (2 points)

(Based on JM, Chapter 3.) Which one of the following two words has a greater edit distance from the word *drives*: *briefs* or *diverse*?

Part 1: Illustrate your calculation by aligning both word pairs.

Part 2: But how do you know these are indeed the best alignments? Therefore, for one of the two pairs, provide the dynamic programming table: that is, the chart that will be gradually filled in by the Min-Edit-Distance algorithm (JM Fig. 3.25, or p. 14 of the February 4 lecture slides).

Part 3: Does this measured similarity (or distance) in these two word pairs correspond to your naive intuition? If not, please explain why and how your intuition about the similarity and distance of words diverge from the formally defined edit distance.

Problem 3: Is *edit distance* a distance metric? (4 points)

Refer to the lecture of February 4 for a formal definition of a *distance metric* (p. 11 of the lecture slides). Show that *edit distance* is a distance metric, at least "under reasonable conditions". What those "reasonable conditions" should be (e.g., regarding the costs of insertion, deletion and substitution)?

The edit distance of two strings is first informally introduced (JM, section 3.11), whereas the Min-Edit-Distance algorithm (JM Fig. 3.25, or p. 14 of the February 4 lecture slides) can also be seen as a definition of edit distance. (If we had a formal definition of edit distance, then we could prove the correctness of the algorithm.) In general, you can refer to the informal "definition" in your proof. However, in order to receive the maximum amount of points, you should base at least some parts of the proof on the algorithm: explain why the algorithm returns a value that satisfies the criteria of a distance metric.

For the mathematically inclined among you, the appendix of this problem set contains an attempt to provide a formal definition. You can also refer to it in your proof.

Appendix

For the mathematically inclined among you, here comes an attempt to provide a formal definition of edit distance. In this self-made definition, I use the + sign for string concatenation, while $\mathbb{R}_0^+$ refers to the set of non-negative real numbers.

Given an alphabet Σ, we shall first introduce an *insertion cost function* $c_i : \Sigma \to \mathbb{R}_0^+$, a *deletion cost function* $c_d : \Sigma \to \mathbb{R}_0^+$, and a *substitution cost function* $c_s : \Sigma \times \Sigma \to \mathbb{R}_0^+$. Then,

Definition 1. *Let a and b be two strings over the alphabet Σ. We say that string a is an* insertion-neighbors *of b if and only if there exist an $x \in \Sigma^*$, $y \in \Sigma^*$ and $\sigma \in \Sigma$ such that $a = x + \sigma + y$ and $b = x + y$.*
Then, let the insertion-distance *of a and b be $D_i(a, b) = c_i(\sigma)$.*

Definition 2. *Let a and b be two strings over the alphabet Σ. We say that string a is a* deletion-neighbors *of b if and only if there exist an $x \in \Sigma^*$, $y \in \Sigma^*$ and $\sigma \in \Sigma$ such that $a = x + y$ and $b = x + \sigma + y$.*
Then, let the deletion-distance *of a and b be $D_d(a, b) = c_d(\sigma)$.*

Definition 3. *Let a and b be two strings over the alphabet Σ. We say that string a is a* substitution-neighbors *of b if and only if there exist an $x \in \Sigma^*$, $y \in \Sigma^*$, $\sigma_1 \in \Sigma$ and $\sigma_2 \in \Sigma$ such that $a = x + \sigma_1 + y$ and $b = x + \sigma_2 + y$.*
Then, let the substitution-distance *of a and b be $D_s(a, b) = c_s(\sigma_1, \sigma_2)$.*

Then, here comes a lemma stating that if two strings are neighbors according to one of the three definitions above, then they cannot be neighbors for the other two definitions (easily proven by referring to the length of the two strings). Therefore, we can move on to the following:

Definition 4. *Let a and b be two strings over the alphabet Σ. We say that a and b are* neighbors *if and only if a is an insertion-neighbor, or a deletion-neighbor, or a substitution-neighbor of b.*
Then, let their distance *be*

$$D(a,b) = \begin{cases} D_i(a,b) & \text{if } a \text{ is an insertion-neighbor of } b \\ D_d(a,b) & \text{if } a \text{ is a deletion-neighbor of } b \\ D_s(a,b) & \text{if } a \text{ is a substitution-neighbor of } b \end{cases}$$

Subsequently,

Definition 5. *Let a and b be two strings over the alphabet Σ. A series of strings $(x_0, x_1, \ldots, x_n)$ over the same alphabet is* an alignment *from a to b if and only if $x_0 = a$, $x_n = b$, while x_i and x_{i+1} are neighbors for all $0 \leq i < n$.*[1]
Let the cost of an alignment *be*

$$C(x_0, x_1, \ldots, x_n) = \sum_{i=0}^{n-1} D(x_i, x_{i+1})$$

[1] If $a = b$, then an alignment from a to b may consist of a single string. Do you get why?

And finally, we can introduce the *edit distance* of any two strings:

Definition 6. *Let a and b be two strings over the alphabet Σ. The* edit distance *of a and b is*

$$D(a,b) = \min\{C(x_0, x_1, \ldots, x_n) | (x_0, x_1, \ldots, x_n) \text{ is an alignment from } a \text{ to } b\}$$