

CIS5210 - Artificial Intelligence - Chapter 5 Notes

Jonathon Delemos - Chris Callison Burch

September 21, 2025

0.1 Big Ideas - Representing Constraints

Constraints must be represented in uniform declarative language. The goal is to find solutions by *general purpose* search algorithms with no changes from problem to problem. It sounds a bit not P vs. NP. where we are treating all problems as P. Our mission is to specify the problem in a formal declarative lagnuage, and a general-purpose algorithm does everything else.

0.2 CSP - Rules

CSP - Consists of:

- **Elements::**

- Finite set of variables
- Non-empty domain of possible values
- Finite set of constraints

- **Character Classes (Disjunction):**

- / [wW]oodchuck/ matches “woodchuck” or “Woodchuck”
- / [abc]/ matches “a”, “b”, or “c”
- / [0-9]/ matches any digit
- / [a-z]/, / [A-Z]/ match lowercase or uppercase letters
- / [b-g]/ matches any character from b to g

- **Negated Character Classes:**

- / \a/ matches any character except “a”

- **Optional Characters (?):**

- ? means the preceding character or nothing
- /woodchucks?/ matches “woodchuck” or “woodchucks”
- /colou?r/ matches “color” or “colour”

- **Repetition (Kleene Star and Plus):**

- /a*/ matches zero or more “a” characters
- /aa*/ matches one or more “a” characters
- / [ab]*/ matches any string of a’s and b’s (including empty)
- / [0-9]+/ matches one or more digits (shorthand for integer)

- **Wildcard (.):**

- /beg.n/ matches “begin”, “beg’n”, “begun”, etc.

- **Anchors and Boundaries:**

- \B matches non-word boundary

- **Grouping and Precedence**

- /cat|dog/ matches cat or dog

- **Special Characters**

- * — Zero or more occurrences of the preceding expression
- + — One or more occurrences of the preceding expression
- ? — Zero or one occurrence (optional) of the preceding expression
- {n} — Exactly n occurrences
- {n,m} — Between n and m occurrences (inclusive)
- {n,} — At least n occurrences
- {,m} — Up to m occurrences

So /a.24z/ will match a followed by 24 dots followed by z (but not a followed by 23 or 25 dots followed by a z).

0.3 2.1.6 - Substitution, Capture Groups, and Eliza

If we want to capture the first match and apply that elsewhere in a string, we can use a **capture group**.

Example:

The (.*)er there were, the /1er they will be.

Here the 1 represents the string we first captured with the .* regular expression. This section is going to be a little confusing. Here we have: /(?:some—a few) (people—cats) like some 1/. This means we ignore the (some — a few) when capturing the expression and instead **match it**. **Eliza works through substitutions**.

0.4 2.2 - Words

Corpus is a computer readable collection of text or speech. We can think of **types** as the vocabulary, and **instances** as the total number N of running words. We can use **Herdan's Law**:

$$|V| = k \cdot N^\beta$$

to calculate for the *types* V. When creating a **datasheet**, it's important to consider these properties:

- Motivation
- Situation
- Language Variety
- Speaker Demographics
- Collection Process
- Annotation Process

- Distribution

For example let's begin with the 'complete words' of Shakespeare in one file, sh.txt. We can use tr to tokenize the words by changing every sequence of non-alphabetic characters to a newline ('A-Za-z' means alphabetic and the -c option complements to non-alphabetic, so together they mean to change every non-alphabetic character into a newline. The -s ('squeeze') option is used to replace the result of multiple consecutive changes into a single output, so a series of non-alphabetic characters in a row would all be 'squeezed' into a single newline): (tr -sc 'A-Za-z' 'n' (\n) >sh.txt)

This creates a **token-per-line** format!

Shall I compare thee to a summer's day?

- Shall
- I
- compare
- thee
- to
- a
- summer
- s
- day

Then we can pass them through a sort, uniq, and uppercase function. Uniq will count the number of uniq words, uppercase will send all words to uppercase, and sort will sort them based off frequency. With that information, sentiment analysis is practically done.

0.5 2.5 - Word and Subword Tokenization

Tokenization is the task of dividing running text into words. The **Byte Pair Encoding** token learner begins BPE with a vocabulary that is just the set of all individual characters. It then examines the training corpus, chooses the two symbols that are most frequently adjacent (say 'A', 'B'), adds a new merged symbol 'AB' to the vocabulary, and replaces every adjacent 'A' 'B' in the corpus with the new 'AB'. It continues to count and merge, creating new longer and longer character strings, until k merges have been done creating k novel tokens; k is thus a parameter of the algorithm. The resulting vocabulary consists of the original set of characters plus k new symbols

Visual Example:

- Input: h e l l o h e l p h e l l
- Step 1: Merge (h, e) → he
- he l l o he l p he l l
- Step 2: Merge (l, l) → ll
- he ll o he l p he ll
- Step 3: Merge (he, ll) → hell
- hell o he l p hell

0.6 2.6 - Word Normalization, Lemmatization, Stemming

Normalization means mapping all input words into a readable form. Example: Sending all words to upper or lower case. **Lemmatization** is the task of determining that two words have the same root. The words am, are, is, have the same shared *be*. Lemmatization is done through **morphological parsing**. Morphology is the study of the way words are build up from smaller meaning bearing units. This is broken down into **stems** and **affixes**. The basic concept behind this is simplifying conjugation down to it's base word.

0.7 Maximum Edit Distance

Edit Distance gives us a way to quantify the minor differences between two strings. Example: coreference and conference. The gap between *intention* and *execution* is 5: delete i, substitute e for n, substitute x for t, insert c, substitute u for n.

How do we find the minimum edit distance?

The space of all possible edits is enormous, so we can't search naively. We must use **dynamic programming**. Dynamic programming is an algorithmic technique used to solve complex problems by breaking them down into simpler, overlapping subproblems.

$D[i, j]$ as the edit distance between $X[1..i]$ and $Y [1.. j]$, i.e., the first i characters of X and the first j characters of Y . The edit distance between X and Y is thus $D[n, m]$. We'll use dynamic programming to compute $D[n, m]$ bottom up, combining solutions to subproblems. In the base case, with a source substring of length i but an empty target string, going from i characters to 0 requires i deletes. With a target substring of length j but an empty source going from 0 characters to j characters requires j inserts. Having computed $D[i, j]$ for small i, j we then compute larger $D[i, j]$ based on previously computed smaller values. The value of $D[i, j]$ is computed by taking the minimum of the three possible paths through the matrix which arrive there:

$$D[i, j] = \min \begin{cases} D[i - 1, j] + \text{del-cost}(\text{source}[i]) \\ D[i, j - 1] + \text{ins-cost}(\text{target}[j]) \\ D[i - 1, j - 1] + \text{sub-cost}(\text{source}[i], \text{target}[j]) \end{cases}$$

You can make it more flexible by assigning higher or lower costs to different substitutions (e.g., for phonetic similarity), but usually it's just 0 or 1.

0.8 Summary

This weeks reading focuses on regular expressions, tokenization, and edit distance. RegEx is a powerful programming language built into linux that allows the user to sift through large text files. Tokenization is concerned with transforming the text into bite size words that the computer can count and evaluate.

0.9 Questions