Compression

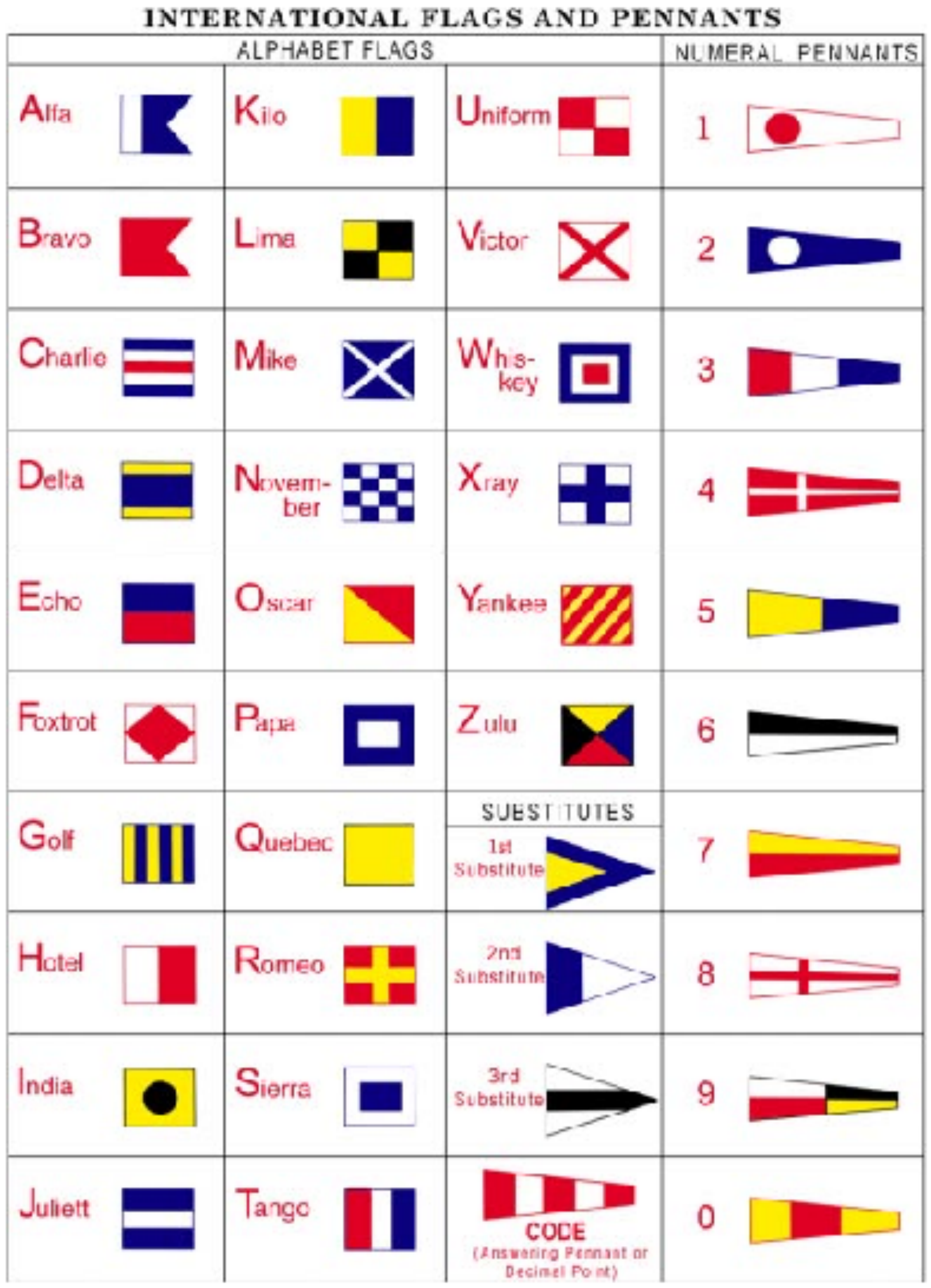
Reducing the space we need

Space = Time

- Any algorithm which traverses a data structure will, if it has to read every element, take time proportional to the size of the structure.
- Because internal memory is so fast, we usually don’t worry too much about this.
- But, what if we have to create a message based on the data? Now, the time to transmit a unit of data (e.g. byte) may be quite long (relatively speaking).
- We may want to compress this message to minimize the time taken to transmit it.

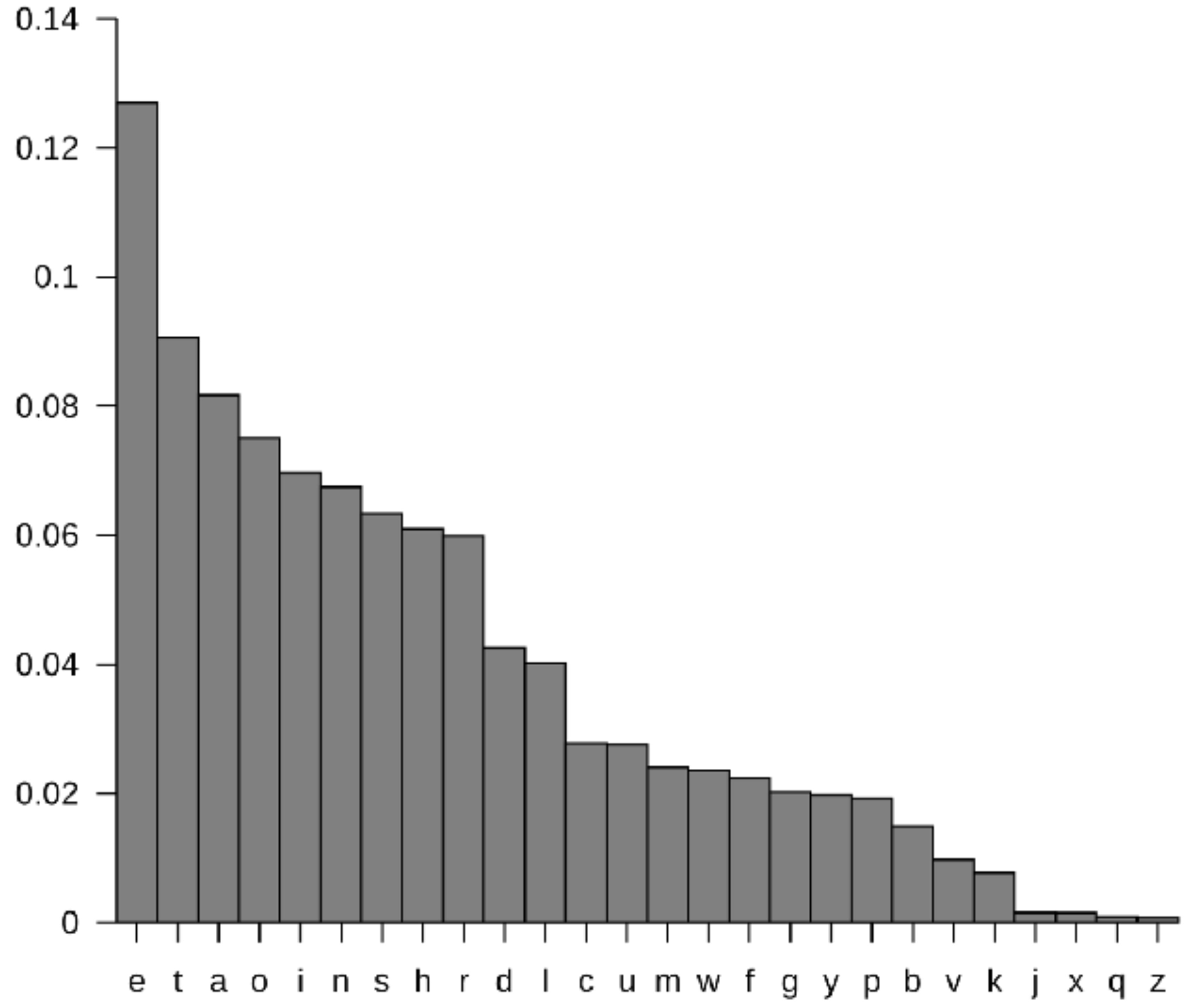
Signalling—Flags

- Naval signalling:
  - Used flags:
- Signals typically included metacharacters for, e.g., repeat, end-of-signal, etc.
- Flags could be used for letter or for fixed phrase:
  - Alfa: Diver down
  - Papa: “Blue Peter”: ship leaving port



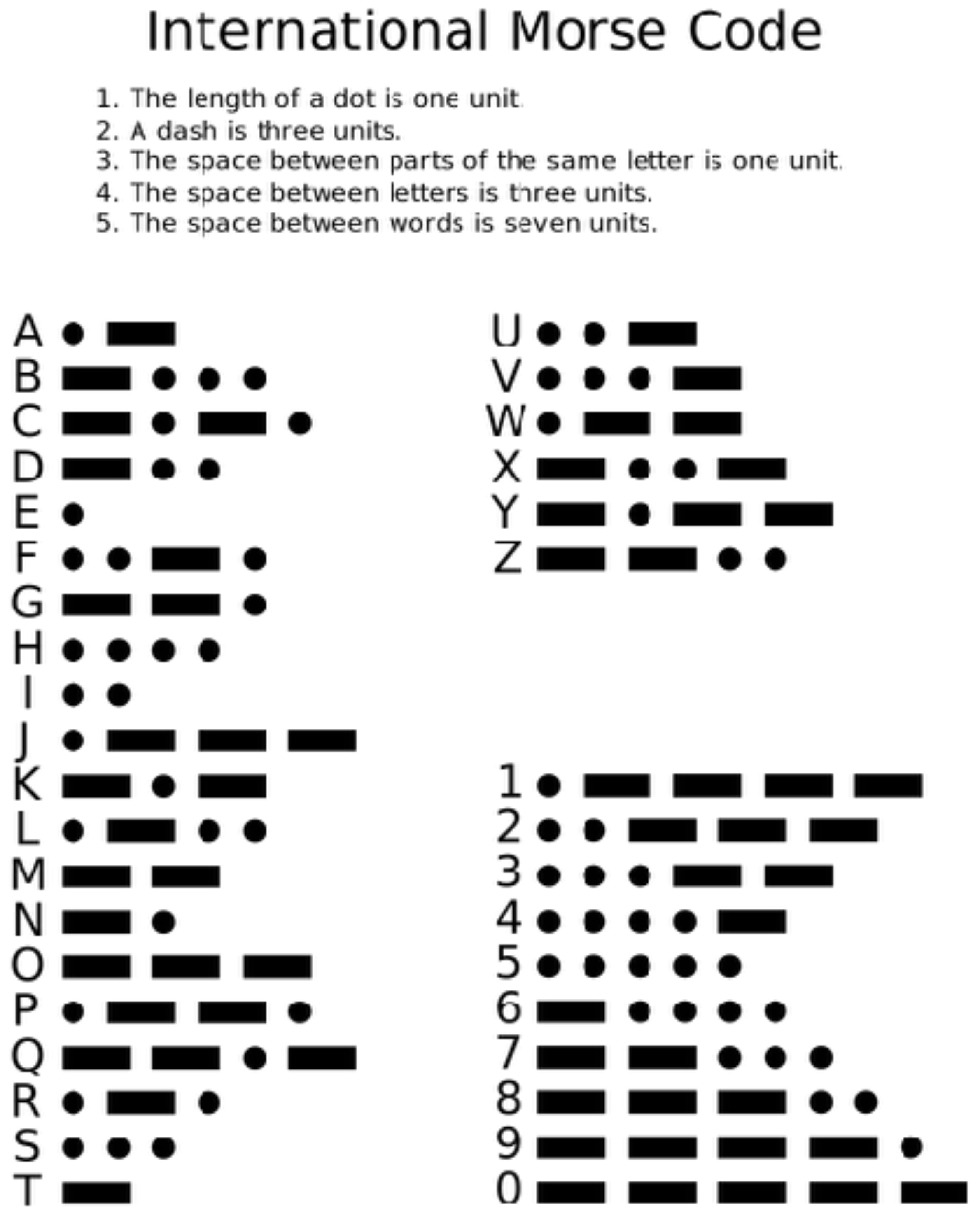
Frequencies of letters in English…

- …Are not constant (true for all languages)
- Wikipedia



Morse code

- Samuel Morse recognized these differences in frequency when he created his telegraph code in 1837.
- Two distinguishable pulses: dot and dash (binary).
- The more frequent letters (e, t, etc.) use fewer pulses.
- The greatest number of pulses per letter is four.

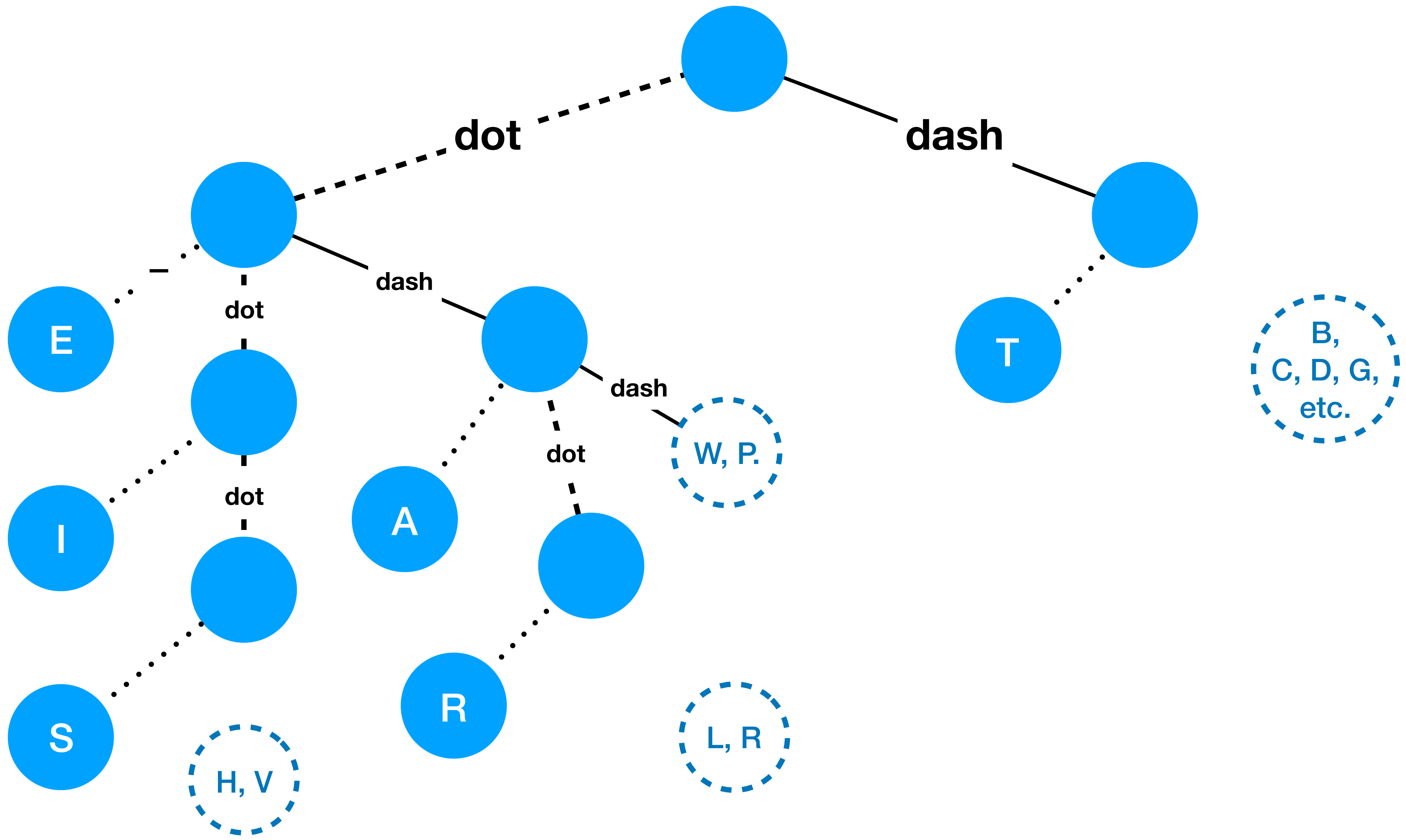


International Morse Code

1. The length of a dot is one unit.
2. A dash is three units.
3. The space between parts of the same letter is one unit.
4. The space between letters is three units.
5. The space between words is seven units.

A	● ▬
B	▬ ● ● ●
C	▬ ● ▬ ●
D	▬ ● ●
E	●
F	● ● ▬ ●
G	▬ ▬ ●
H	● ● ● ●
I	● ●
J	● ▬ ▬ ▬
K	▬ ● ▬
L	● ▬ ● ●
M	▬ ▬
N	▬ ●
O	▬ ▬ ▬
P	● ▬ ▬ ●
Q	▬ ▬ ● ▬
R	● ▬ ●
S	● ● ●
T	▬
U	● ● ▬
V	● ● ● ▬
W	● ▬ ▬
X	▬ ● ● ▬
Y	▬ ● ▬ ▬
Z	▬ ▬ ● ●
1	● ▬ ▬ ▬ ▬
2	● ● ▬ ▬ ▬
3	● ● ● ▬ ▬
4	● ● ● ● ▬
5	● ● ● ● ●
6	▬ ● ● ● ●
7	▬ ▬ ● ● ●
8	▬ ▬ ▬ ● ●
9	▬ ▬ ▬ ▬ ●
0	▬ ▬ ▬ ▬ ▬

Morse Code as a Decision Tree



Huffman Codes

- There is one inefficiency present in Morse code: it’s a ternary (not binary) code—because there are empty time intervals between letters—or stop codes.
- How can we use the notion of frequency distribution to minimize the average length of a message using binary digits only?
- David Huffman wondered this in 1962. He devised a simple binary tree such that all leaf nodes are characters and that, on average, the message length is minimized.

Huffman Coding (1)

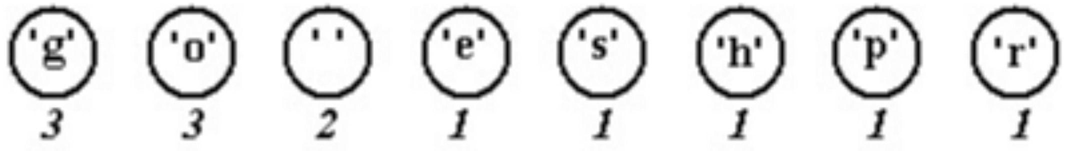
- So, what sort of algorithm can provide us this tree?
    - When we are coding a single character, we descend the tree until we reach the leaf (terminal) node that matches that character.
    - If *d(x)* is the depth of *x*, and *p(x)* is the probability of the next character being *x*, then the sum of all the terms *d(x) p(x)* should be a minimum (because it is the expected message length).
    - Therefore, at each node, we want to have equal probability of descending into the left tree as into the right tree. That’s to say the total frequencies of all leaf nodes on the left should equal (approximately) the total frequencies of all leaf nodes on the right.

Huffman Coding (2)

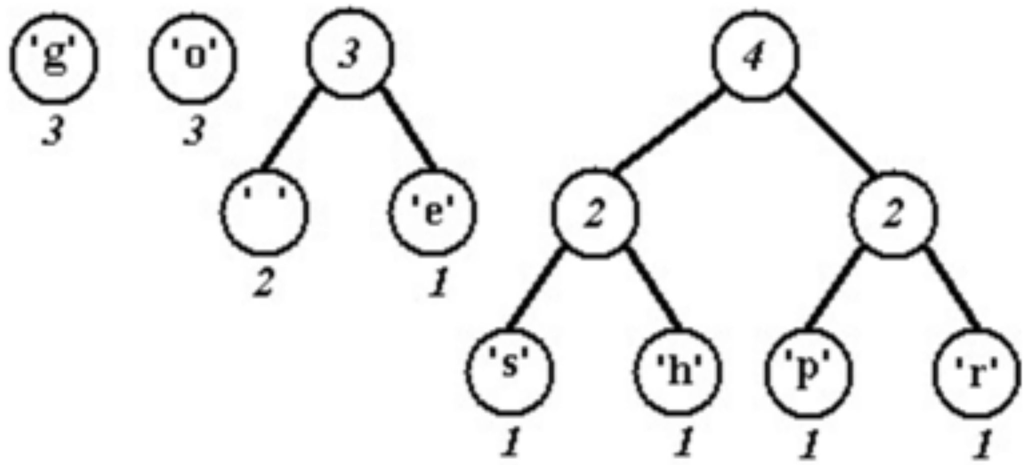
- So, let’s start out with a forest of *N* single-node trees, each representing a different character. Each node has a property: weight—that’s to say the sum of the frequencies of the leaves under this node (including the node itself). Only leaf nodes have the additional property of a character.
- Then, we iterate through *N-1* steps, combing the two nodes with the lowest weights into a binary tree. Obviously, the weight of the new parent node is the sum of w_l and w_r where w_l and w_r are the weights of the left and right trees, respectively.
- Example: an alphabet of the following characters only: g, o, ‘ ‘, e, s, h, p, r; with probabilities as follows:
  - *p(g), p(o)* = 3
  - *p(‘ ‘)* = 2
  - *p(e), p(s), p(h), p(p), p(r)* = 1
- We have these specific probabilities because the only message we wish to send is *go go gophers*. [The gophers are the Minnesota Uni football team.]

Huffman Coding (3)

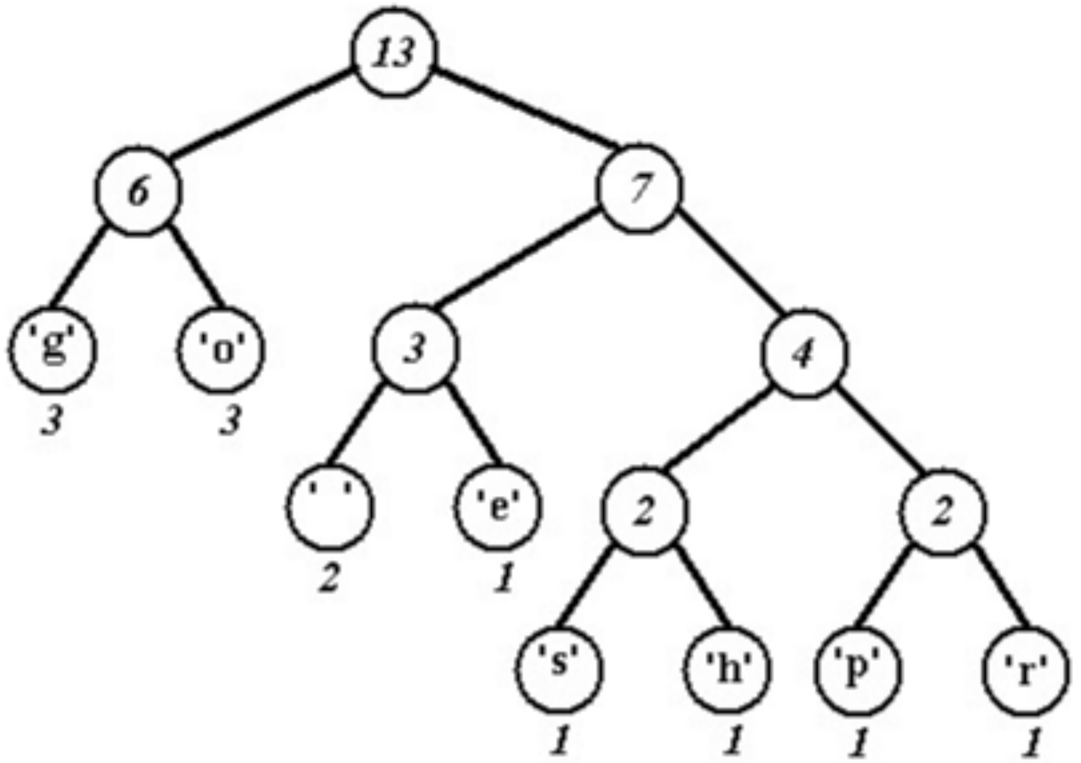
Initial set up



After 4 iterations



After 7 iterations

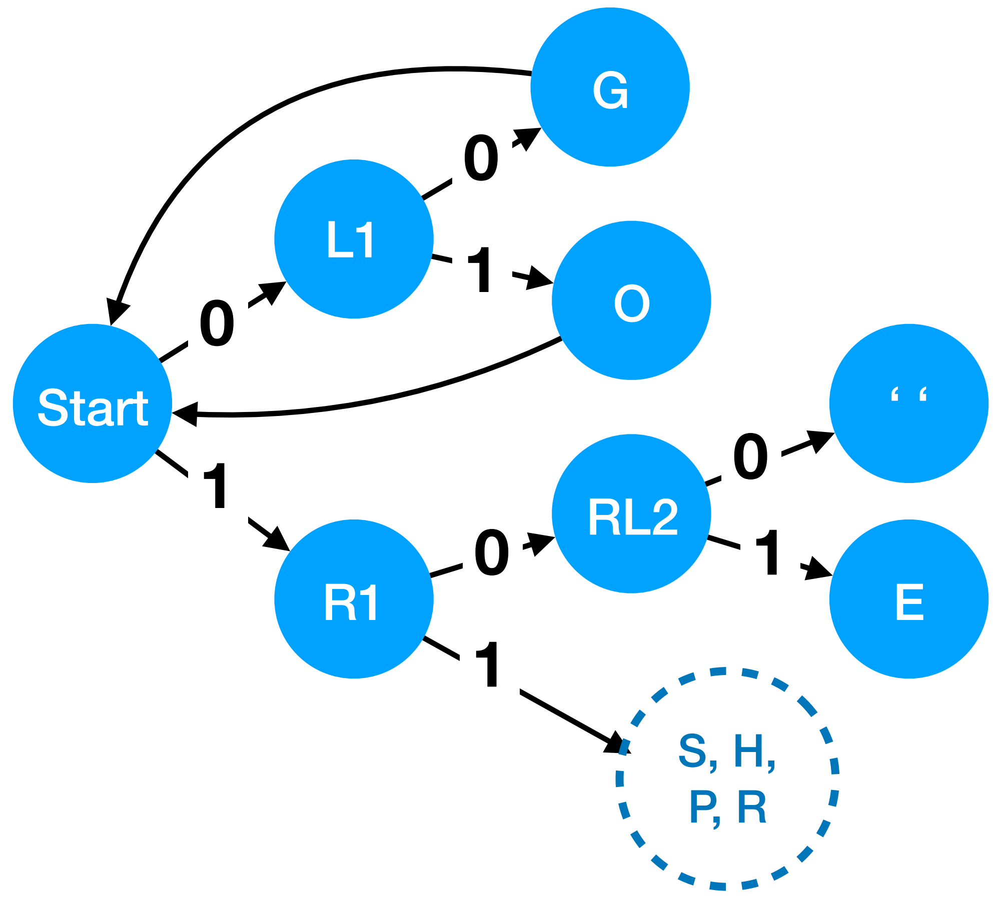


Huffman Coding (4)

- Now, we use the tree we created for coding messages.
- The only message we want to send is *go go gophers*.
- We generate the message by choosing a 0 for traversing the left branch and a 1 for traversing the right branch.
- Our message thus looks like this:
    - 00 01 100 00 01 100 00 01 1110 1101 101 1111 1100
- We show gaps between the characters for our own visualization, but realize that these gaps don't exist in the message itself.
- We used 37 bits to encode this message. If we had used ASCII 7-bit code, we would have needed 13x7 = 91 bits.
- See articles: Wikipedia, Go go gophers, beep boop beer.

Huffman Coding (5)

- So, we have received a message which is the following: 000110000011000001111011011011111100
- How do we decode it?
    - For that, we need a finite state machine. It's basically just the same old tree but usually represented on its side—all end-states (characters) return the state to the start (shown for "G", "O"):



Huffman Coding (6)

- Huffman coding uses a “greedy” algorithm. Why?
    - Remember, we arbitrarily chose the two smallest weights and combined them at each iteration. That is a localized decision which doesn’t take into account the global minimum, nor does it allow backtracking.

Run-length encoding

- Suppose we have a rasterized image (the result of scanning):
  - We have, for example, 1000 rows of 1000 pixels. Each pixel has a color which is either black or white (this is used for sending a “fax”, something we used to do all the time!
  - The probability of adjacent pixels in a row being the same (black or white) is quite high.
  - Therefore, when we change color, we simply say how many pixels there are in this “run” (its length).
  - Really simple! It’s much harder to use coherence in the vertical direction (between scan lines) although it is possible.
  - And, if these images represent a black and white TV image, then there may be *z*-coherence too (*z* is the time domain).

Run-length encoding (2)

- For a 1024-pixel row, we have the probability that a pixel is black is 1/4. Assuming that we use an 8-bit integer for each run, how many bits will we need for our 1024 pixel row on average?
    - Assuming that we always start with a white run (possibly of zero-length) and alternate colors between runs. The probability of each length of run is as follows:
        - 0: 0.25 (when the row starts with a black pixel, or after a 256-length run)
        - 1: 0.25 x 0.75 (black run) + 0.75 x 0.25 (white run) = 0.375
        - 2: 0.125 x 0.75 (black) + 0.875 x 0.25 (white) = 0.3125
        - 3: 0.0625 x 0.75 (black) + 0.9375 x 0.25 (white) = 0.28125
        - etc.
    - Mean run-length will be something like 2.5 so we should expect to need