Searching for Strings

Almost every computer application that deals with text has a command that will search for strings.

- on OS/X and Windows systems, the command is typically called Find
- located in the Edit menu

Example

I connected to NCBI to look at the genome file for Yersinia pestis bacterium that causes plague.

It's a big file: 11,388,115 characters

4.6 million characters in the file are DNA letters

Example (cont’d)

Searching for “trna” (note this search isn’t case sensitive):

4
Another Example

- I downloaded the file to my system and opened it up with **emacs**, a text editor widely used by programmers on Unix systems
- emacs has an "incremental search" that shows results as you type the characters
- also shows other matches in the window

![Emacs screenshot](image)

Exact Match

- The “find” command in the web browser and the “incremental search” command in emacs are examples of substring searches
- They implement an **exact match** algorithm
- **input**: a body of text $T$ and a pattern $P$
- **output**: locations in $T$ where letters of $T$ are exactly the same as letters in $P$
- **Example:**

  $P$  
  pick
  $T$  
  peter piper picked a peck of pickled peppers

![Exact Match Diagram](image)

Text as a String

- In this view of the problem, the text is just one long string object
- **In Ruby:**
  ```ruby
  >> p = "pick"
  >> t = "peter piper picked a peck of pickled peppers"
  >> i = t.index(p)
  => 12
  >> j = i + p.length - 1
  => 15
  >> t[i..j]
  => "pick"
  ```

Remember the first character is $t[0]$, so "pick" starts at $t[12]$

Text as a String (cont’d)

- Using Ruby to find “trna” in **Yersinia pestis**
  ```ruby
  >> t = IO.read("NC_004088.gbk"); nil
  => nil
  >> t.length
  => 11388115
  >> t.index("trna")
  => nil
  >> t.index("tRNA")
  => 17756
  ```

Ruby evaluates an expression of the form $x : y$ by evaluating both $x$ and $y$ and returning the value of $y$

- **Notes:**
  - Ruby's index method is **case sensitive**
  - nil is a special object that means "nothing"
  - It’s a good value to return from a string search (or other algorithm) that doesn’t find what it’s looking for
Notation

- In the remaining slides we’ll look at three algorithms that solve the exact match problem.
- Common notation used in all the algorithms:
  - $T$ is the text to search
    - $n$ letters in $T$
    - $i$ is an index (value from 0 to $n-1$)
  - $P$ is the pattern to look for
    - $m$ letters in $P$
    - $j$ is an index (value from 0 to $m-1$)

Simple String Search

- The simplest algorithm to search for a substring scans from left to right
- think of a “window” the size of the pattern sliding over the text
- at each step of the algorithm compare the two strings in the window

```
for i = 0 to n-m
  for j = 0 to m-1
    if $T_{i+j}$ ≠ $P_j$
      match = false
      break
  if match
    return i
return nil
```

Note:
- break means “exit the current loop”
- if match is still true when the inner loop exits then we found all the letters in $P$

Analysis of Simple Search

- This search algorithm has nested loops
  - for $i$ ← 0 to $n-m$
    - for $j$ ← 0 to $m-1$
      - very similar in overall structure to insertion sort and matching pairs
- The worst case number of steps is when
  - the pattern is not in the text
  - the inner loop makes many “false starts”
  - example: search for “abcd” in “abcabababcaab...”
- This algorithm is $O(n \times m)$
  - in practice, when searching for short patterns in diverse text, it’s effectively $O(n)$
Automaton

- String searching can be done by an automaton created from the pattern
- Put the text you want to scan on the input tape and start the machine
- If the machine is in the final state when the tape has been read the pattern was found somewhere in the text
- To build the FSA:
  - make one state for each position in the pattern
  - link the states with forward-pointing transitions labeled by the letters of the pattern

![Initial FSA for the pattern “anagram”](image)

Failure Transitions

- Next figure out what to do if the machine gets part way through the pattern
- example: input is “bang”
- after reading “an” the machine is in state 2
- the next letter (“g”) should cause the machine to go back to the start state
- Add failure transitions that take the FSA back to the beginning on a mismatch:

![Failure Transitions](image)

Failure Transitions (cont’d)

- These new transitions are not the right course of action in every case, however
- example: suppose the input starts out “baa…”
- the second “a” sends the machine back to state 0
- that means it will fail to find the pattern in “baanagrams”

![Failure Transitions (cont’d)](image)
Failure Transitions (cont’d)

- This new machine still has a problem, though
- suppose the input is “bananagrams”
- this new machine gets to state 3 after “bana”, and the second “n” causes it to go back to state 0
- the remaining input is “agrams” and the FSA doesn’t reach state 7

![Diagram of FSA States]

Failure Transitions (Finally)

- The final solution has links that remember the machine has seen “an”
- if a state was reached because the machine read an “a” and the next letter is an “n” go back to state 2
- there are two places where this can happen in our FSA
- the new transitions are shown in green

![Updated Diagram of FSA States]

Finite State Machine Method

- Summary of the FSA search method:
  - build an FSA based on the pattern
    - a straightforward (but not trivial) algorithm examines all cases for failure transitions
    - scan the text, keeping track of which state the machine is in
  - Is this better than the simple search shown earlier?
    - pro: each letter of the text is read once
    - algorithm runs in $O(n)$ steps ($n$ is the length of the text)
    - simple search requires $O(n \times m)$ ($m$ is the length of the pattern)
    - con: it takes time to build the FSA and extra space to store it
  - Note: if an application uses a pattern more than once it can reuse the FSA
    - build the machine once
    - use it any number of times on new input texts

FSA in Ruby

- What does it mean to “build” an FSA?
- In Ruby we could use a hash for the transitions of each state
  - key: input letter
  - value: next state id
- Examples:
  - state 0: `{ “a” => 1 }`
  - state 1: `{ “a” => 1, “n” => 2 }`
- Collect all the hashes into an array:
  ```ruby
  delta = [
    { “a” => 1 },
    { “a” => 1, “n” => 2 },
    ...
  ]
  ```
  - An array of Hash objects
The Boyer-Moore Algorithm

- An algorithm defined by R. Boyer and J. S. Moore in 1977 use the “sliding window” approach of the simple algorithm, but with a small difference:

  * scan the window from right to left

  Peter piper picked a peck of pickled peppers

  The window starts at the left edge of the text, as in the simple algorithm
  Start scanning on the right side of the window

Boyer-Moore (cont’d)

- When we find a mismatch, we can ask whether the text letter occurs anywhere in the pattern:
  - if not, the window can be moved past the mismatch
  - slide the window an amount equal to the length of the pattern
- In this example, “e” does not occur anywhere in the pattern, so we can slide the window over four places:

  Peter piper picked a peck of pickled peppers
  Peter piper picked a peck of pickled peppers
  Peter piper picked a peck of pickled peppers
  The Boyer-Moore algorithm is able to skip 3 places that are compared with the simple algorithm

Boyer-Moore (cont’d)

- In the next step, there is again a mismatch in the first comparison
  - but this time the letter “i” does occur in the pattern
  - we can’t move the window 4 places, but we can move it so the two i’s line up

  Peter piper picked a peck of pickled peppers

  Peter piper picked a peck of pickled peppers

  Now we have a mismatch with another “e” and the window can move 4 places again:

  Peter piper picked a peck of pickled peppers

Preprocessing

- As in the automata method, the Boyer-Moore algorithm uses tables of information that are built by scanning the pattern before the search starts
  - the algorithm uses two tables
  - they are very similar to the “failure” links in the automaton
  - one tells how far to slide the window based on letters in the pattern
  - the other takes into account partial matches (like the “an” prefix in “anagram”)
- When an algorithm does some initial work before the main loop it is known as preprocessing
The Boyer-Moore algorithm is very simple once the tables are built:

\[
\text{lambda} = \text{last-occurrences}(P) \\
\text{gamma} = \text{good-suffixes}(P) \\
i \leftarrow 0 \\
\text{while } i < n-m \\
j \leftarrow m-1 \\
\text{while } j \geq 0 \text{ and } T_{i+j} \neq P_j \\
j \leftarrow j-1 \\
\text{if } j = -1 \\
\text{return } i \\
\text{else} \\
i \leftarrow i + \max(\text{gamma}[j], j-\text{lambda}[T_{i+j}]) \\
\text{return } \text{nil}
\]

Note:
- \(\text{lambda}[x]\) is the amount to shift the window when there is a mismatch with \(x\)
- \(\text{gamma}[n]\) is the amount to shift based on a matching suffix of \(P\)

Review

A simple algorithm for string search slides a “window” along the text
- the size of the window is the length of the pattern
- compare pattern to text left to right inside the window
- move window one place right when a mismatch is found
- takes \(O(n \times m)\) steps

One can create an FSA to do a string search
- \(m + 1\) states
- forward links represent matches
- backward links are taken on mismatches
  - not every mismatch restarts from state 0 -- remember prefixes
- takes \(O(n)\) steps after preprocessing to build the FSA