Comb 555 - BioAlgorithms - Spring 2021

Combinatorial Pattern Matching

- Problem set #2 is due tonight
A Recurring Problem

- Finding patterns within sequences

- Variants on this idea
  - Finding repeated motifs amongst a set of strings
  - What are the most frequent k-mers
  - How many times does a specific k-mer appear

- Fundamental problem: *Pattern Matching*
  - Find all positions of a particular substring in given sequence?
The most fundamental of pattern matching problems-- does a pattern, $p$, appear in a text, $t$? And if so, where?

- **Goal:** Find all occurrences of a pattern in a text
- **Input:** Pattern $p = p_1, p_2, \ldots, p_n$ and text $t = t_1, t_2, \ldots, t_m$
- **Output:** All positions $1 < i < (m - n + 1)$ such that the $n$-letter substring of $t$ starting at $i$ matches $p$

```python
In [2]: def bruteForcePatternMatching(p, t):
    locations = []
    for i in range(0, len(t)-len(p)+1):
        if t[i:i+len(p)] == p:
            locations.append(i)
    return locations

print(bruteForcePatternMatching("ssi", "imissmissmississippi"))
```

[11, 14]
Pattern Matching Performance

- Performance:
  - $m$ - length of the text $t$
  - $n$ - the length of the pattern $p$
  - Search Loop - executed $O(m)$ times
  - Comparison - $O(n)$ symbols compared
  - Total cost - $O(mn)$ per pattern

- In practice, most comparisons will terminate early. Why?

- But worst-case data examples exist:
  - $p = "AAAT"
  - $t = "AAAAAAAAAAAAAAAAAAAAAAAAAAAT"$
We can do better!

If we preprocess our pattern we can search more efficiently ($O(n)$).

Example: FindPattern("ssi", "imissmissmississippi"):

```
   imissmissmississippi
1.       s
2.       s
3.       s
4.       SSi
5.       s
6.       SSi
7.       s
8.       SSI   - match at 11
9.       SSI   - match at 14
10.        s
11.        s
12.        s
```

- At steps 4 and 6 after finding the mismatch "i" ≠ "m" we can skip over all positions tested because we know that the suffix "sm" is not a prefix of our pattern "ssi".
- Even works for our worst-case example "AAAAT" in "AAAAAAAAAAAAAAAAAT" by recognizing the shared prefixes ("AAA" in "AAAA").
- How about finding multiple patterns $[p_1,p_2,...,p_3]$ in $t$
Keyword Trees

- We can preprocess the set of strings we are seeking to minimize the number of comparisons.

  - **Idea:** Combine patterns that share prefixes, to share those comparisons.
    - Stores a set of keywords in a rooted labeled tree.
    - Each edge labeled with a letter from an alphabet.
    - All edges leaving a given vertex have distinct labels.
    - Leaf vertices are indicated.
    - Every keyword stored can be spelled on a path from the root to some leaf vertex.
    - Searches are performed by “threading” the target pattern through the tree.

- A **Tree** is a special graph as discussed previously.
  - One connected component.
  - \( N \) nodes, \( N-1 \) edges, No loops.
  - Exactly one path from any.

- A **Trie** is a tree that is related to a sequence.
  - Generally, there is a 1-to-1 correspondence between either nodes or edges of the trie and a symbol of the sequence.
Prefix Trie Match

- **Input**: A text $t$ and a trie $P$ of patterns
- **Output**: True if $t$ leads to a leaf in $P$; False otherwise

What is output for:
- apple
- band
- april

Performance:
- $O(m)$ - the length of the text, $t$
- Independent of how many strings are in the Keyword Trie
Prefix Trie code

```
In [2]:
def path(string, parent):
    if (len(string) > 0):
        if (string[0] in parent):
            child = parent[string[0]]
        else:
            child = {}
            parent[string[0]] = child
            path(string[1:], child)
    else:
        parent['$'] = True

class PrefixTrie:
    def __init__(self):
        """ Tree is a dictionary of the children at each node""
        self.root = {}
    def add(self, string):
        """ Add a path from the Trie's root""
        path(string, self.root)
    def match(self, string):
        """ Check if there is a path from the root to a '$' ""
        parent = self.root
        for c in string:
            if c not in parent:
                break
            parent = parent[c]
        else:
            return '$' in parent
        return False
```
```python
In [3]:
T = PrefixTrie()
T.add("apple")
T.add("banana")
T.add("apricot")
T.add("bandana")
T.add("orange")
print(T.root)
print(T.match("orange"))

print([T.match(v) for v in ['apple', 'banana', 'apricot', 'orange', 'band', 'april', 'bandana', 'bananapple']])

{'o': {'r': {'a': {'n': {'g': {'e': {'$': True}}}}}}},
{'b': {'a': {'n': {'a': {'n': {'a': {'$': True}}}}}},
{'d': {'a': {'n': {'a': {'$': True}}}}}},
{'a': {'p': {'r': {'i': {'c': {'o': {'t': {'$': True}}}}}}},
{'p': {'l': {'e': {'$': True}}}}}}
True
[True, True, True, False, False, True, False]
Multiple Pattern Matching

Suppose that we have a long string, $t$, like a genome, and we want to find if any of the strings in a previously constructed prefix trie, $P$, appear within it.

- $t$ - the text to search through
- $P$ - the trie of patterns to search for

```python
def multiplePatternMatching(t, P):
    locations = []
    for i in xrange(0, len(t)):
        if PrefixTrieMatch(t[i:], P):
            locations.append(i)
    return locations
```
Multiple Pattern Matching Example

multiplePatternMatching("bananapple", P):
  0: PrefixTrieMatching("bananapple", P) = True
  1: PrefixTrieMatching("ananapple", P) = False
  2: PrefixTrieMatching("nanapple", P) = False
  3: PrefixTrieMatching("anapple", P) = False
  4: PrefixTrieMatching("napple", P) = False
  5: PrefixTrieMatching("apple", P) = True
  6: PrefixTrieMatching("pple", P) = False
  7: PrefixTrieMatching("ple", P) = False
  8: PrefixTrieMatching("le", P) = False
  9: PrefixTrieMatching("e", P) = False

locations = [0, 5]
Trie Improvements

- Based on our previous speed-up
- We can add failure edges to our Trie
  Add an edge to any prefix from the root that matches a suffix on our failed path
- Aho-Corasick Algorithm

The concept of "threading" one string through another

bapple
bap
apple
Multiple Pattern Matching Performance

- $m \cdot \text{len}(t)$
- $d$ - max depth of $P$ (longest pattern in $P$)
- $O(md)$ to find all patterns
- Can be decreased further to $O(m)$ using Aho-Corasick Algorithm
  - Add links for pattern suffixes that match text prefixes
- Pattern matching data structure is query specific

**Idea:** Rather than building a search data structure for indexing the prefixes of the pattern, why not build one for indexing the suffixes of the text.
Now for a Twist

● What if our list of keywords were simply all suffixes of a **single given string**

Example: ATCATG
  TCATG
  CATG
  ATG
  TG
  G

● The resulting keyword tree:
  A Suffix Trie

● How would you find "CAT"
  ● It is a prefix of one of our suffixes
  ● If there is a path for our entire pattern, we know which suffix it came from
  ● Try "AT"
A *compressed* Suffix Trie

- Combine nodes with in and out degree 1
- Make edges of these substrings
- All internal nodes have at least 3 edges
- All leaf nodes are labeled with an index of the suffix's index
Uses for Suffix Trees

- Suffix trees hold all suffixes of a text, T
  - i.e., ATCATG: ATCATG, TCATG, CATG, ATG, TG, G
- Can be built in $O(m)$ time for text of length $m$
- To find any pattern $P$ in a text:
  - Build suffix tree for text, $O(m)$, $m=|T|$
  - Thread the pattern through the suffix tree
  - Can find pattern in $O(n)$ time! ($n=|P|$)
- $O(|T|+|P|)$ time for "Pattern Matching Problem" (better than Naïve $O(|P||T|)$)
- Build suffix tree and lookup pattern
- Multiple Pattern Matching in $O(|T|+k|P|)$
Suffix Tree Overhead

- Input: text of length $m$
- Computation
  - $O(m)$ to compute a suffix tree
  - Does not require building the suffix trie first
- Memory
  - $O(m)$ - nodes are stored as offsets and lengths
- Huge hidden constant, best implementations
- Requires about $20 \times m$ bytes
- 3 GB human genome = 60 GB RAM
Suffix Tree Examples

● What is the string represented in the suffix tree? Find path that leads to "1"

● What letter occurs most frequently? Find edge from the root leads to the most leafs

● How many times does "ATG" appear, and where? Match "ATG" to tree and count the number of leafs from that path

● How long is the longest repeated k-mer? Find longest path leading to two leafs
Suffix Trees: Theory vs. Practice

- In theory, suffix trees are extremely powerful for making a variety of queries concerning a sequence
  - What is the shortest unique substring?
  - How many times does a given string appear in a text?
- Despite the existence of linear-time construction algorithms, and $O(m)$ search times, suffix trees are still rarely used for genome-scale searching
- Large storage overhead
### Substring Searching

- Is there some other data structure to gain efficient access to all of the suffixes of a given string with less overhead than a suffix tree?

- Some things we know
  - Searching an unordered list of items with length $n$ generally requires $O(n)$ steps
  - However, if we sort our items first, then we can search using $O(log(n))$ steps
  - Thus, if we plan to do frequent searches there is some advantage to performing a sort first and amortizing its cost over many searches

- For strings suffixes are interesting items. Why?

| Suffixes: panamabanananas anamabanananas namabanananas amabanananas mabananas abanananas bananas ananas nanas anas nas as s | Sorted Suffixes: abanananas amabanananas anamabanananas ananas anas as bananas mabananas namabanananas nanas nas nas panamabanananas s |
Questions you can ask

Is there any use for a list of sorted suffixes?

**Sorted Suffixes:**
- abananas
- amabananas
- anamabananas
- ananas
- anas
- as
- bananas
- mabananas
- namabananas
- nanas
- nas
- panamabananas
- s

- Does the substring "nana" appear in the original string?
- How many times does "ana" appear in the string?
- What is the most/least frequent letter in the original string?
- What is the most frequent two-letter substring in the original string?
Properties of a sorted “suffix array”

- Size of the sorted list if the given text has a length of $m$? $O(m^2)$
- Cost of the sort? $O(m^2 \log(m))$
- Not practical for big $m$
- There are many ways to sort
  - What is an “in place” sort?
  - What is a “stable” sort?
  - What is an “arg” sort?
Arg Sorting

consider the list:

\[ [72, 27, 45, 36, 18, 54, 9, 63] \]

When sorted it is simply:

\[ [9, 18, 27, 36, 45, 54, 63, 72] \]

Its “arg” sort is:

\[ [6, 4, 1, 3, 2, 5, 7, 0] \]

- The \textit{ith} element in the arg sort is the \textit{index} of the \textit{ith} element from the original list when sorted.
- Thus, \([A[i] \text{ for } i \text{ in } \text{argsort}(A)] == \text{sorted}(A)\)
Code for Arg Sorting

```python
In [7]:

def argsort(input):
    return sorted(range(len(input)), key=input.__getitem__)

A = [72, 27, 45, 36, 18, 54, 9, 63]
print(argsort(A))
print([A[i] for i in argsort(A)])

print()
B = ['TAGACAT', 'AGACAT', 'GACAT', 'ACAT', 'CAT', 'AT', 'T']
print(argsort(B))
print([B[i] for i in argsort(B)])

[6, 4, 1, 3, 2, 5, 7, 0]
[9, 18, 27, 36, 45, 54, 63, 72]

[3, 1, 5, 4, 2, 6, 0]
Next Time

- We'll see how arg sorting can be used to simplify representing our sorted list of suffixes
- Suffix arrays
- Burrows-Wheeler Transforms
- Applications in sequence alignment