CMPUT 403: Strings

Zachary Friggstad

March 11, 2016

Outline

- Tries
- Suffix Arrays
- Knuth-Morris-Pratt Pattern Matching

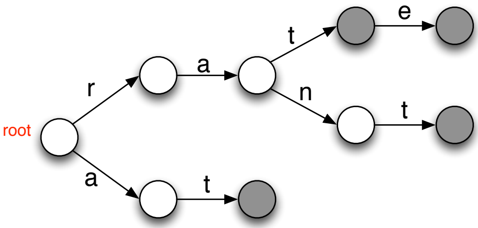
Tries

Given a dictionary D of strings and a query string s, determine if s is in D.

Using a trie, this can be done in $O(|s|)$ time after some preprocessing that takes $O(T)$ time and space where T is the total size of D.

Example

A trie for dictionary {rat, rate, rant, at}.



A trie is simply a tree whose children can be quickly indexed by characters.

Each node also stores a boolean value indicating it is a *terminal* node (i.e. it represents a string in D). Can store other useful info too.

```
struct TrieNode {
  bool word;
  unordered_map<char, TrieNode> child;
  Trie() : word(false) {}
};

//initializing :)
TrieNode root;
```

Building the Trie

Add each dictionary string, one at a time.

To add a string, just go as far in the Trie with the string as possible and then start creating new nodes.

```
void add_string(TrieNode* root, const string& s) {
  //constructs a new TrieNode if the child didn't exist
  for (auto c : s) root = &root->child[c];
  root->word = true; //record this string
}

vector<string> dict;
for (const auto& s : dict) add_string(&root, s);
```

Adding one string s takes $O(s)$ time.
So adding all strings takes $O(T)$ time.

Querying

Given a trie representing a dictionary D and given a string s, is $s \in D$?

```
bool query(TrieNode* root, const string& s) {
  for (auto c : s) {
      auto it = root->child.find(c);
      //no child is indexed by c
      if (it == root->child.end()) return false;
      root = &it->second;
  }
  //done traversing, check that we ended at a terminal
  return root->word;
}
```

Other Query Examples

How many strings in D begin with s? Need to keep an int at each node v storing # of strings represented below v.

Suffix Arrays

Sort all of the *suffixes* of a string lexicographically.

`bananaban`

- `aban`
- `an`
- `anaban`
- `ananaban`
- `ban`
- `bananaban`
- `n`
- `naban`
- `nanaban`

Obvious how to do it: $O(n^2 \log n)$ - create all suffixes and sort them.

Can actually get $O(n)$ time!

This is overkill and a bit technical, let's see an $O(n \log^2 n)$ algorithm.

Suffix Array

An array of indices of the start positions of the suffixes in sorted order.

Example

For string `bananaban`

```
int sarray[] = {5, 7, 3, 1, 6, 0, 8, 4, 2};
```

Idea: For $i = 0, \ldots, \log_2 n$, sort the suffixes just by their first 2^i characters.

- ananaban
- anaban
- aban
- an
- bananaban
- ban
- nanaban
- naban
- n

Can do in $O(n \log n)$ time (recall we are actually just sorting the indices, not the whole suffixes).

Next, sort the suffixes by their length 2 prefixes.

- `aban`
- `ananaban`
- `anaban`
- `an`

- `bananaban`
- `ban`

- `n`
- `nanaban`
- `naban`

Next, sort the suffixes by their length 4 prefixes.

- aban

- an
- anaban
- ananaban

- ban
- bananaban

- n

- naban
- nanaban

To check if nanaban $<$ naban, just look up the 2nd half of the red parts to see how they were ordered last step.

Generally, to sort the suffixes by their length 2^{i+1} prefixes we check $<$ using the ordering based on length 2^i prefixes.

Check how the first 2^i characters of two suffixes a, b compare using the previous ordering. If they are different then just return that result.

If they are the same, check how the second 2^i characters of a, b compare again using the previous ordering.

Example

`nanaban` vs. `naban`. Length-2 prefixes are the same (`na`), but next 2 characters (`na` vs. `ba`) show the answer is $>$.

Sorting based on length 2^i prefixes then takes only $O(n \log n)$ time. Since i ranges up to $\log_2 n$, then overall time is $O(n \log^2 n)$.

Can also quickly compute the longest common prefix between adjacent suffixes in the array.

Example

`naban` and `nanaban` are adjacent suffixes in the suffix array.

Their common prefix length is 2. This information can easily be construct along with the construction of the suffix array itself.

Knuth-Morris-Pratt

Given a source string s and a pattern string p, does p appear as a substring of a?

More generally, record all positions i such that p appears as a substring of a starting at position i.

Example

`s = findmatchingmatches`

`p = match`

Then p appears as a substring of s at indices 4 and 12 (highlighted).

An obvious algorithm is to try all locations of s and linearly scan to see if p matches there.

Can take $\Omega(|s| \cdot |p|)$ time. The Knuth-Morris-Pratt (KMP) algorithm only takes $O(|s| + |p|)$ time!

Main Idea: For each index i into p, let $\pi[i]$ denote the length of the longest proper suffix of $p_0 p_1 \ldots p_i$ that is also a prefix of p.

Confusing? Example!

`p = acabaca`

The longest proper suffix of `acabac` that is also a prefix is `ac`.

`pi = {0, 0, 1, 0, 1, 2, 3};`

Slide the pattern p "over" s.

```
acabaca
acacabacabaca
```

Match as many symbols as possible

```
acabaca
acacabacabaca
```

When stuck, slide the pattern to the next partial match.

```
  acabaca
acacabacabaca
```

Distance to slide encoded by prefix table π.

Continue matching

```
  acabaca
acacabacabaca
```

Found a match, record it! Slide pattern over to the next partial match.

```
      acabaca
acacabacabaca
```

Continue matching

```
      acabaca
acacabacabaca
```

Another match, record it!

Slide pattern over to next partial match.

```
          acabaca
acacabacabaca
```

Quit, the pattern is past the end of the string.

Runs in $O(|s| + |p|)$ time because each step increases the "matched" pointer or slides the pattern over. Each slide takes $O(1)$ time using the π values.

```
void kmp(const string& s, const string& p) {
    vector<int> pi;
    compute_prefix(p, pi); //next two slides :)

    //hit = # matched, i = position of pattern
    int hit = 0, i = 0;

    while (i + p.length() < s.length()) {
        //match as much as possible
        while (hit < p.length() && p[hit] == s[i + hit]) ++hit;

        //record a match
        if (hit == p.length()) cout << "Match:" << i << endl;

        //shift pattern
        i += hit - pi[hit];
        hit = pi[hit];
    }
}
```

How to compute π? Basically the same idea!

`p = acabaca`

Note, a suffix of $\pi[i]$ that is also a prefix of p comes from a suffix of $\pi[i-1]$ that is a prefix of p.

`acabac`: Note `a` is a suffix of `acaba` that is also a prefix of p.

Overall idea: slide the pattern over itself!

```
    acabaca
acabaca
```

When computing $\pi[i]$, if the current "slide" matches then nothing to do (see above).

Otherwise shift pattern over until there is a match.

```
void compute_prefix(const string& p, vector<int>& pi) {
  pi.resize(p.length());
  pi[0] = 0;

  for (int i = 1, hit = 0; i < p.length(); ++i) {
  while (i < p.length()) {
    while (hit > 0 && p[hit] != p[i]) hit = pi[hit];
    if (p[hit] != p[i] ) pi[i] = 0; //hit == 0 here
    else p[i] = ++hit;
  }
}
```

Missing Topics

- Suffix Trees
- Manachar's Algorithm: find all *maximal palindromes* in linear time.

Next Week

Bipartite matching and network flow.

Remember to vote on your preferred *extra* topics for the weeks following!