String Algorithms and Data Structures

Tries

CS 199-225
Brad Solomon

October 3, 2022
Exact pattern matching \textit{w/} Boyer-Moore

\textbf{As seen in HW:} sub-linear time \textit{in practice}

\[ \text{Boyer-Moore} \approx O(|P| + |T|) \]

Make lookup tables

Find instances of \( P \) in \( T \)
Preprocessing: Live chat streams

**Patterns:** banned phrases

**Text:** Chat messages
Preprocessing: Live chat streams

- Find instances of $P$ in $T$

Amortize cost of preprocessing $P$ over many $T$

- Preprocess
- Boyer-Moore
- Thanks for the help!
- I don’t understand that…
- You are a !@#$%! teacher
Exact pattern matching with indexing

Conventionally $T \gg P$:

$P_1 P_2$
$P_3 P_4$

Preprocess (index)

Search Index

Find instances of $P$ in $T$

*Amortize* cost of preprocessing $T$ over many $P$
Preprocessing: Libraries

Patterns: Book of interest

Text: All books in library
Preprocessing: Libraries

Preprocess the library by indexing all the books.
Preprocessing: Libraries

List of all library books

Preprocess (index)

Search Index

Find instances of $P$ in $T$

Given full library, built an index once* that is re-used
Exact pattern matching \( w/ \) indexing

Find instances of \( P \) in \( T \)

Preprocess (index) \( \approx O(|T|) \)

Search Index \( \approx O(|P|) \)

What information from \( T \) do we need to search for \( P \)?
Preprocessing for exact pattern matching

$T$: CGTGC

$P$: 

Search($P, T$):

$P$: 

Search($P, T$):

$P$: 

Search($P, T$):
Preprocessing for exact pattern matching

A substring $S$  

The position of $S$ in $T$
Preprocessing for exact pattern matching

$T$: CGTGC

$|T|$ | $|T-1|$ | $|T-2|$ |

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>0</td>
</tr>
<tr>
<td>G</td>
<td>1</td>
</tr>
<tr>
<td>T</td>
<td>2</td>
</tr>
<tr>
<td>G</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>4</td>
</tr>
<tr>
<td>CG</td>
<td>0</td>
</tr>
<tr>
<td>GT</td>
<td>1</td>
</tr>
<tr>
<td>TG</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Preprocessing for exact pattern matching

$T$: C G T G C

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>0</td>
</tr>
<tr>
<td>G</td>
<td>1</td>
</tr>
<tr>
<td>T</td>
<td>2</td>
</tr>
<tr>
<td>G</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>4</td>
</tr>
<tr>
<td>CG</td>
<td>0</td>
</tr>
<tr>
<td>GT</td>
<td>1</td>
</tr>
<tr>
<td>TG</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

$\frac{|T|(|T|+1)}{2}$
Preprocessing for exact pattern matching

Because our keys are strings, this is sometimes possible!

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>0</td>
</tr>
<tr>
<td>G</td>
<td>1</td>
</tr>
<tr>
<td>T</td>
<td>2</td>
</tr>
<tr>
<td>G</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>4</td>
</tr>
<tr>
<td>CG</td>
<td>0</td>
</tr>
<tr>
<td>GT</td>
<td>1</td>
</tr>
<tr>
<td>TG</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

We want to search in $O(|P|)$ without $O(|T|^2)$ space!
Preprocessing for exact pattern matching

Strings consist of individual characters!

... and these characters can overlap:

\[ T: C G T G C \]

![Diagram showing preprocessing for exact pattern matching]
Preprocessing for exact pattern matching

Strings consist of individual characters!

... and these characters can overlap:

$T: C \ G \ T \ G \ C$

G
GT
GTG
GTGC

root:

G
T
G
C

...
Preprocessing for exact pattern matching

Strings consist of individual characters!

… and these characters can overlap:

$T:\overset{T}{C}G\overset{T}{T}G\overset{C}{C}$

Diagram:

```
root: G -> T -> G -> C
    ↓     ↓     ↓     ↓
  C       G       T       G
  ↓       ↓       ↓       ↓
T       TG       TGC
```
Preprocessing for exact pattern matching

Strings consist of individual characters!

... and these characters can overlap:

$T: C\ G\ T\ G\ C$

![Diagram of a tree with nodes labeled with characters and edges connecting them. The tree starts with a root node labeled with G, and branches out with edges labeled with T, G, C, and G.]
Preprocessing for exact pattern matching

Strings consist of individual characters!

... and these characters can overlap:

$T: C G T G C$

\[
\text{root: } \quad \begin{array}{c}
G \\
T \\
G \\
C
\end{array} \quad \begin{array}{c}
G \\
T \\
G \\
C
\end{array} \quad \begin{array}{c}
G \\
C
\end{array}
\]
String indexing with Tries

**Trie:** A rooted tree storing a collection of (key, value) pairs

Keys:  

- `instant`  
- `internal`  
- `internet`

Values:  

- 1  
- 2  
- 3

Each edge is labeled with a character $c \in \Sigma$

For given node, at most one child edge has label $c$, for any $c \in \Sigma$

Each key is “spelled out” along some path starting at root
String indexing with Tries

**Trie:** A rooted tree storing a collection of (key, value) pairs

Keys:      Values:

instant    1
internal   2
internet   3

Each edge is labeled with a character $c \in \Sigma$

For given node, at most one child edge has label $c$, for any $c \in \Sigma$

Each key is “spelled out” along some path starting at root
String indexing with Tries

**Trie**: A rooted tree storing a collection of (key, value) pairs

<table>
<thead>
<tr>
<th>Keys</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>instant</td>
<td>1</td>
</tr>
<tr>
<td>internal</td>
<td>2</td>
</tr>
<tr>
<td>internet</td>
<td>3</td>
</tr>
</tbody>
</table>

Where should I store the value 1?
String indexing with Tries

**Trie:** A rooted tree storing a collection of (key, value) pairs

Keys:          Values:

instant 1
internal 2
internet 3
internets 4

Each key’s value is stored at the last node in the path
The Node Implementation

Each node in my trie has $\leq |\Sigma|$ edges!

Each edge is a (potentially NULL) pointer.

How can we encode this?
The Node Implementation

Each node in my trie has \( \leq |\Sigma| \) edges!

Each edge is a (potentially NULL) pointer.

1) Static Array

2) Dynamically-sized Dictionary (std::map)
class NaryTree
{
    public:
        struct Node {
            std::vector<int> index;
            std::map<char, Node*> children;

            Node(std::string s, int i)
            {
                if(s.length() > 0 ){
                    children[s[0]] = new Node(s.substr(1), i);
                } else {
                    index.push_back(i);
                }
            }
        };

    protected:
        Node* root;
};
class NaryTree
{
    public:
        struct Node {
            std::vector<int> index;
            std::map<char, Node*> children;

            Node(std::string s, int i)
            {
                if(s.length() > 0 ){
                    children[s[0]] = new Node(s.substr(1), i);
                } else {
                    index.push_back(i);
                }
            }

            protected:
                Node* root;
        }
}
main.cpp

1 NaryTree myT;
2 myTree.print();
3 myTree.insert("AB",0);
4 myTree.print();
5 myTree.insert("ABA",1);
6 myTree.print();
7 myTree.insert("ABB",2);
8 myTree.print();
9 myTree.insert("BAB",3);
10 myTree.print();
11 myTree.insert("BBB",4);
12 myTree.print();

Trie Node Implementation
main.cpp

NaryTree myT;
myTree.print();

myTree.insert("AB",0);
myTree.print();

myTree.insert("ABA",1);
myTree.print();

myTree.insert("ABB",2);
myTree.print();

myTree.insert("BAB",3);
myTree.print();

myTree.insert("BBB",4);
myTree.print();

Trie Node Implementation
Trie Node Implementation

```cpp
main.cpp

NaryTree myT;
myTree.print();
myTree.insert("AB",0);
myTree.print();
myTree.insert("ABA",1);
myTree.print();
myTree.insert("ABB",2);
myTree.print();
myTree.insert("BAB",3);
myTree.print();
myTree.insert("BBB",4);
myTree.print();
```

Former leaf node, still holds value

```
struct Node {
    std::vector<int> index;
    std::map<char, Node*> children;
}
```
Trie Node Implementation

```cpp
#include <iostream>

using namespace std;

class MyTree { // NaryTree
    public:
        int insert(string str, int length);
        void print();

    private:
        int level1[26];
        int level2[26][26];
        string tree[26][26];
        int len;

    public:
        MyTree();

    ~MyTree();

    MyTree* child[26];

    void print() {
        cout << endl;
    }

    int insert(string str, int length) {
        int current = 0;
        for (int i = 0; i < length; i++) {
            int index = str[i] - 'A';
            if (child[index] == NULL) {
                child[index] = new MyTree();
            }
            current = child[index];
            if (current->tree.length() <= length) {
                current->tree[str[i] - 'A'] = str[i];
                current->level2[index][length - current->len] = str[i];
                current->level1[index] = str[i];
                current->len = length;
            }
            current = current->child[index];
        }
        return current->len;
    }

    // Constructor
    MyTree() {
        for (int i = 0; i < 26; i++) {
            child[i] = NULL;
            for (int j = 0; j < 26; j++) {
                level1[i] = 0;
                level2[i][j] = 0;
            }
        }
    }

    // Destructor
    ~MyTree() {
        for (int i = 0; i < 26; i++) {
            delete child[i];
        }
    }

};

int main() {
    MyTree myT;
    myTree.print();
    myTree.insert("AB", 0);
    myTree.print();
    myTree.insert("ABA", 1);
    myTree.print();
    myTree.insert("ABB", 2);
    myTree.print();
    myTree.insert("BAB", 3);
    myTree.print();
    myTree.insert("BBB", 4);
    myTree.print();
};
```
Trie Node Implementation

```cpp
main.cpp

NaryTree myT;
myTree.print();
myTree.insert("AB",0);
myTree.print();
myTree.insert("ABA",1);
myTree.print();
myTree.insert("ABB",2);
myTree.print();
myTree.insert("BAB",3);
myTree.print();
myTree.insert("BBB",4);
myTree.print();
```

```
             o
            /|
           / | 0
          /  |
         /   |
        /    |
       /     |
      /      |
     /       |
    /        |
   /         |
  +---------+--
  |         |
  |         |
  +--------+
```

```
             o
            /|
           / | 1
          /  |
         /   |
        /    |
       /     |
      /      |
     /       |
    /        |
   /         |
  +---------+--
  |         |
  |         |
  +--------+
```
Trie Node Implementation

```
main.cpp

NaryTree myT;
myTree.print();
myTree.insert("AB",0);
myTree.print();
myTree.insert("ABA",1);
myTree.print();
myTree.insert("ABB",2);
myTree.print();
myTree.insert("BAB",3);
myTree.print();
myTree.insert("BBB",4);
myTree.print();
```

```
  o
 +--A--o  0
 |     +--B--o  1
 |         +--A--o
|             +--B--o
+--B--o  2

 o
 +--A--o
 |     +--B--o
 |         +--A--o
|             +--B--o
+--B--o

 o
 +--A--o
 |     +--B--o
 |         +--A--o
|             +--B--o
+--B--o

 o
 +--A--o
 |     +--B--o
 |         +--B--o
+--B--o

```

Trie Node Implementation
Trie Node Implementation

```cpp
void NaryTree::insert(const std::string& s, int i)
{
    insert(root, s, i);
}

void NaryTree::insert(Node*& node, const std::string & s, int i)
{
    // If we're at a NULL pointer, we make a new Node
    if (node == NULL) {
        node = new Node(s, i);
    } else {
        if(s.length() > 0 ){
            if(node->children.count(s[0]) > 0){
                insert(node->children[s[0]],s.substr(1), i);
            }else{
                node->children[s[0]] = new Node(s.substr(1), i);
            }
        } else{
            node->index.push_back(i);
        }
    }
}
```
Assignment 5: a_narytree

Learning Objective:

Store all substrings in a trie using NaryTree implementation

Implement exact pattern matching using this trie

Consider: How many insertions are we doing for each string? Is there a better or faster way to do this?
Trie Node Implementation

```
main.cpp

NaryTree myT;
myTree.insert("AB",0);
myTree.insert("AB",2);
myTree.print();
```

```
o
+-A-+    ??
 |   |
+-B-+  
```

```cpp
NaryTree myT;
myTree.insert("AB",0);
myTree.insert("AB",2);
myTree.print();
```

```
struct Node {
    std::vector<int> index;
    std::map<char, Node*> children;
}
```

```
if(s.length() > 0 ){
    if(node->children.count(s[0]) > 0){
        insert(node->children[s[0]],s.substr(1), i);
    }else{
        node->children[s[0]] = new Node(s.substr(1), i);
    }
} else{
    node->index.push_back(i);
}
```
Searching a Trie

Given $P$, search the trie for keys and return values

Pattern: infer

1. Try to match front character
2. If match, move to appropriate child
   2.5 Set pattern equal to remainder
   2.5 Go back to (1)
3. If mismatch, $P$ is not a key!

Let's break that down using recursion:

Starting at root:

1. Try to match front character
2. If match, move to appropriate child
3. If mismatch, $P$ is not a key!
Searching a Trie

Given $P$, search the trie for keys and return values

Pattern: $i n t e r e s t i n g$

0. $i n t e r e s t i n g$

Lets break that down using recursion:

Starting at root:

1. Try to match front character

2. If match, move to appropriate child

   2.5. Set pattern equal to remainder

   2.5. Go back to (1)

3. If mismatch, $P$ is not a key!
Searching a Trie

Given $P$, search the trie for keys and return values

Pattern: instra

Lets break that down using *recursion*:

Starting at root:

1. Try to match front character
2. If match, move to appropriate child
   
   (2.5) Set pattern equal to remainder
   
   (2.5) Go back to (1)
3. If mismatch, $P$ is not a key!
Searching a Trie

Given $P$, search the trie for keys and return values

Pattern: \texttt{insta insta}

Let's break that down using \textit{recursion}:

Starting at root:

1. Try to match front character
2. If match, move to appropriate child
   - 2.5 Set pattern equal to remainder
   - 2.5 Go back to (1)
3. If mismatch, $P$ is not a key!

"\texttt{Insta}" is NOT a key! There's no value here!
String indexing with Tries

A rooted tree storing a collection of (key, value) pairs

Keys:               Values:

\[ \text{instant} \quad 1 \]
\[ \text{internal} \quad 2 \]
\[ \text{internet} \quad 3 \]

The trie is structured such that:

- Each edge is labeled with a character \( c \in \Sigma \)
- For given node, at most one child edge has label \( c \), for any \( c \in \Sigma \)
- Each key is “spelled out” along some path starting at root
- Each key’s value is stored at the last node in the path
Searching a Trie

Given $P$, search the trie for keys and return values

Pattern: insta

Let's break that down using recursion:

Starting at root:

1. Try to match front character
2. If match, move to appropriate child
   2.5 Set pattern equal to remainder
   2.5 Go back to 1
3. If mismatch, $P$ is not a key!

"Insta" is NOT a key!
There's no value here!
Searching a Trie

Given $P$, search the trie for keys and return values

Pattern: `instant`

```plaintext
  instant
```

Let's break that down using *recursion*:

Starting at root:

1. Try to match front character
2. If match, move to appropriate child
   2.5 Set pattern equal to remainder
   2.5 Go back to (1)
3. If mismatch, $P$ is not a key!

(0) If we have no ‘front’ char, check value
   0.5 If no value, $P$ is not a key.
   0.5 If value, $P$ is a key, return value(s).
Assignment 5: a_narytree

Learning Objective:

- Store all substrings in a trie using NaryTree implementation
- Implement exact pattern matching using this trie

Consider: How could we search the trie if we are only allowed to store one value in each node [instead of a vector of them]?
Preprocessing for exact pattern matching

\( T: C \ G \ T \ G \ C \)

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>0</td>
</tr>
<tr>
<td>G</td>
<td>1</td>
</tr>
<tr>
<td>T</td>
<td>2</td>
</tr>
<tr>
<td>G</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>4</td>
</tr>
<tr>
<td>CG</td>
<td>0</td>
</tr>
<tr>
<td>GT</td>
<td>1</td>
</tr>
<tr>
<td>TG</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**Key**

- C1
- G1
- T2
- G3
- C4
- root: 0

**Value**

- C1: 0
- G1: 1
- T2: 2
- G3: 3
- C4: 4
- root: 0
Preprocessing for exact pattern matching

We can do exact pattern matching in $O(P)$ time!
Preprocessing for exact pattern matching

\[ T: C G T G C \]

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>0</td>
</tr>
<tr>
<td>G</td>
<td>1</td>
</tr>
<tr>
<td>T</td>
<td>2</td>
</tr>
<tr>
<td>G</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>4</td>
</tr>
<tr>
<td>CG</td>
<td>0</td>
</tr>
<tr>
<td>GT</td>
<td>1</td>
</tr>
<tr>
<td>TG</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

We are storing \( \frac{|T|(|T| + 1)}{2} \) values.

We had to do \( \frac{|T|(|T| + 1)}{2} \) insertions.
Preprocessing for exact pattern matching

If only there was a way...

to insert fewer strings
to store fewer values
Preprocessing for exact pattern matching

If only there was a way...

to insert fewer strings

to store fewer values

to be even more efficient!

$T = \text{abaaba}$