A_zalg reflection

Learning Objectives met

Want better labeling of values

Want better intuition of how / why values work and are needed
A_bmoore due today!

Did you successfully implement preprocessing in linear time?
Exact pattern matching with Boyer-Moore

As seen in HW: sub-linear time in practice

Preprocess

Find instances of $P$ in $T$

Boyer-Moore $\approx O(|P| + |T|)$

Make lookup tables
Exact pattern matching with indexing

Conventionally $T \gg P$:

Find instances of $P$ in $T$

$P_1 P_2$
$P_3 P_4$

Preprocess (index)

Search Index

Amortize cost of preprocessing $T$ over many $P$
Preprocessing: Live chat streams

Patterns: banned phrases  
Text: Chat messages
Preprocessing: Live chat streams

- Preprocess
- Boyer-Moore
- Find instances of $P$ in $T$

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

Patterns: Book of interest

Text: All books in library
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
### Preprocessing: Glossaries

**What method of preprocessing is this?**

Patterns: Key terms

<table>
<thead>
<tr>
<th>Term</th>
<th>Page References</th>
</tr>
</thead>
<tbody>
<tr>
<td>Myrmecocystus</td>
<td>106–9</td>
</tr>
<tr>
<td>exocrine glands</td>
<td></td>
</tr>
<tr>
<td>Hymenoptera</td>
<td>194, 262, 324, 388</td>
</tr>
<tr>
<td>Hypoponera</td>
<td>20–22</td>
</tr>
<tr>
<td>inclusive fitness</td>
<td>20–23, 29–42</td>
</tr>
<tr>
<td>intercastes</td>
<td>388–89</td>
</tr>
<tr>
<td>Iridomyrmex</td>
<td>266, 280, 288, 321</td>
</tr>
<tr>
<td>Isoptera, see termites</td>
<td></td>
</tr>
<tr>
<td>juvenile hormone, caste</td>
<td>106–9, 372</td>
</tr>
<tr>
<td>kin recognition</td>
<td>293–98</td>
</tr>
<tr>
<td>kin selection</td>
<td>18–19, 23–24, 28–42, 299, 386</td>
</tr>
<tr>
<td>Macrotermes (termites)</td>
<td>59–60</td>
</tr>
<tr>
<td>male recognition</td>
<td>298</td>
</tr>
<tr>
<td>mass communication</td>
<td>62–63, 214–18</td>
</tr>
<tr>
<td>mating, multiple</td>
<td>155</td>
</tr>
<tr>
<td>maze following</td>
<td>119</td>
</tr>
<tr>
<td>Megalomyrmex (ants)</td>
<td>457</td>
</tr>
<tr>
<td>Megaponera (ants), see Pachycondyla</td>
<td></td>
</tr>
<tr>
<td>Melipona (stingless bees)</td>
<td>129</td>
</tr>
<tr>
<td>Melophorus (ants), repletes</td>
<td>257</td>
</tr>
<tr>
<td>memory</td>
<td>117–19, 213</td>
</tr>
<tr>
<td>Messor (harvester ants)</td>
<td>212, 232</td>
</tr>
<tr>
<td>mind</td>
<td>117–19</td>
</tr>
<tr>
<td>Monomorium</td>
<td>127, 212, 214, 216–17, 292</td>
</tr>
<tr>
<td>motor displays</td>
<td>235–47</td>
</tr>
<tr>
<td>mound-building ants</td>
<td>2</td>
</tr>
<tr>
<td>multilevel selection</td>
<td>7, 7–13, 24–29</td>
</tr>
<tr>
<td>mutilation, ritual</td>
<td>366–73</td>
</tr>
<tr>
<td>mutualism, see symbioses, ants</td>
<td></td>
</tr>
<tr>
<td>Myrmecyna (fossil ants)</td>
<td>318</td>
</tr>
<tr>
<td>Myrmecia (ants)</td>
<td>326</td>
</tr>
</tbody>
</table>

Text: All text in the book
Preprocessing: Glossaries

Find instances of $P$ in $T$

Glossary built on total contents $T$, useful for multiple $P$
Exact pattern matching \textit{w/ indexing}

1 Billion reads ($\sim$100)

The human genome ($\sim$3 billion)

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

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

Find instances of $P$ in $T$

Amortize cost of preprocessing $T$ over many $P$
Exact pattern matching with indexing

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

$T$: C G T G C

$P$: 

Search($P, T$):

$P$: 

Search($P, T$):

$P$: 

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

T: CGTGC

A substring S

The position of S in T
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>

$|T|$ $|T-1|$ $|T-2|$
Preprocessing for exact pattern matching

\[ T: \text{CGTGC} \]

<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>

\[ \frac{|T|(|T|+1)}{2} \]
Because our keys are strings, this is sometimes possible!

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:
Preprocessing for exact pattern matching

Strings consist of individual characters!

... and these characters can overlap:

$T$: C G T G C

G
G T
G T G
G T G C

root:

G → C → G → T → G → C
Preprocessing for exact pattern matching

Strings consist of individual characters!

... and these characters can overlap:

$T: C G T G C$

$T$

$T G$

$T G C$

root: $G$ $T$ $G$ $C$

Diagram showing the pattern matching process with overlapping characters.
Preprocessing for exact pattern matching

Strings consist of individual characters!

… and these characters can overlap:

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

![Diagram of string matching]
Preprocessing for exact pattern matching

Strings consist of individual characters!

... and these characters can overlap:

$T: C G T G C$

![Tree diagram](image)
String indexing with Tries

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

Keys: \( \text{instant, internal, internet} \)

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:**
- `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

[Diagram of a trie tree with keys `instant`, `internal`, `internet` and values `1`, `2`, `3`.
String indexing with Tries

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

Keys:  Values:

instant  1  
internal  2  
internet  3  

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

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

Keys: 
- instant 1
- internal 2
- internet 3
- internets 4

Values: 
- 1
- 2
- 3
- 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;
    ...
}
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();
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

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

```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

```
+-A--o
  +--A--o
    +--B--o
      +--A--o
```

```
```
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();
```
Trie Node Implementation

```cpp
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();
13
```

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();
```
Trie Node Implementation

NaryTree.h

```cpp
void NaryTree::insert(const std::string& s, int i)
{
    insert(root, s, int 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 = i;
        }
    }
}
```
Assignment 5: a_narytree

Learning Objective:

Store all substrings in a trie using NaryTree implementation

Implement exact pattern matching using this trie

Due: February 28th 11:59 PM

Consider: How many insertions are we doing for each string? Is there a better or faster way to do this?
main.cpp

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

```cpp
main.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
  infer
  infer
  infer

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: interesting
  interesting
  interesting

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
Searching a Trie

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

Pattern: \texttt{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!
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!

"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:
  instant    1
  internal   2
  internet   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: \( i n s t a \)

```
  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!

```
Pattern:
```

\( \text{insta} \)

(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).

(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

Lets break that down using recursion:

Starting at root:

(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).

(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!
Assignment 5: a_narytree

Learning Objective:

Store all substrings in a trie using NaryTree implementation

Implement exact pattern matching using this trie

Due: February 28th 11:59 PM

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

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

<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>

Diagram: 

```
root: G
   / \ 0
  C  T
  |  / \ 1
  |  G  C
  |  / \ 1
  |  T  G
  |  / \ 2
  |  G  C
  |  / \ 1
  |  C  0
```

Key: Value

- C: 0
- G: 1
- T: 2
- G: 3
- C: 4
- CG: 0
- GT: 1
- TG: 2
- ...: ...
Preprocessing for exact pattern matching

\[ T: \text{CGTGC} \]

<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>
</tbody>
</table>

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

\[ T: C G T G C \]

### Key | Value
--- | ---
\( C \) | 0
\( G \) | 1
\( T \) | 2
\( G \) | 3
\( C \) | 4
\( CG \) | 0
\( GT \) | 1
\( TG \) | 2
... | ...

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!