Cache-Oblivious String Dictionaries

Gerth Stølting Brodal
University of Aarhus

Joint work with Rolf Fagerberg

ADS 2005 – 2nd Bertinoro Workshop on Algorithms and Data Structures
29 May-4 June 2005, University of Bologna Residential Center, Bertinoro (Forli), Italy
Outline of Talk

- Cache-oblivious model
- Basic cache-oblivious techniques
- Cache-oblivious string algorithms
  - Cache-oblivious tries and blind tries
Hierarchical Memory Models
Hierarchical Memory

Increasing access time and space

Cache-Oblivious String Dictionaries
Hierarchical Memory

<table>
<thead>
<tr>
<th></th>
<th>Latency</th>
<th>Relative to CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>0.5 ns</td>
<td>1</td>
</tr>
<tr>
<td>L1 cache</td>
<td>0.5 ns</td>
<td>1-2</td>
</tr>
<tr>
<td>L2 cache</td>
<td>3 ns</td>
<td>2-7</td>
</tr>
<tr>
<td>DRAM</td>
<td>150 ns</td>
<td>80-200</td>
</tr>
<tr>
<td>TLB</td>
<td>500+ ns</td>
<td>200-2000</td>
</tr>
<tr>
<td>Disk</td>
<td>10 ms</td>
<td>$10^7$</td>
</tr>
</tbody>
</table>

- Accessing non-local storage may take a very long time
- Good locality is important for achieving high performance
I/O Model

Aggarwal and Vitter 1988

\[ N = \text{problem size} \]
\[ M = \text{memory size} \]
\[ B = \text{I/O block size} \]

- One I/O moves \( B \) consecutive records from/to disk
- Complexity measure = number of I/Os
Ideal Cache Model — no parameters!?

Frigo, Leiserson, Prokop, Ramachandran 1999

- Program with only one memory
- Analyze in the I/O model for
- Optimal off-line cache replacement strategy arbitrary $B$ and $M$
Ideal Cache Model — no parameters!?

Frigo, Leiserson, Prokop, Ramachandran 1999

- Program with only one memory
- Analyze in the I/O model for
- Optimal off-line cache replacement strategy arbitrary $B$ and $M$

Advantages

- Optimal on arbitrary level $\Rightarrow$ optimal on all levels
- Portability, $B$ and $M$ not hard-wired into algorithm
- Dynamic changing $M$ (and $B$)
Cache-Oblivious Preliminaries
Cache-Oblivious Scanning

Corollary

Cache-oblivious selection requires $O\left(\frac{N}{B}\right)$ I/Os

Hoare 1961 / Blum et al. 1973
Cache-Oblivious Scanning

\[ O\left(\frac{N}{B}\right) \text{ I/Os} \]

**Corollary**  Cache-oblivious selection requires \( O(N/B) \) I/Os

Hoare 1961 / Blum et al. 1973
Cache-Aware B-trees

$O(\log_B N)$

Search path
Static Cache-Oblivious B-Tree

Recursive layout of binary tree $\equiv$ van Emde Boas layout
Each green tree has height between \( \log_2 B \) and \( \log_2 B \), i.e., perform at most \( 4 \log_2 B \) I/Os (misalignment).
Static Cache-Oblivious B-Tree

Each green tree has height between $(\log_2 B) = 2$ and $\log_2 B$.

Searches visit between $\log B N$ and $2 \log B N$ green trees, i.e. perform at most $4 \log B N$ I/Os (misalignment).
Each green tree has height between $(\log_2 B)^2$ and $\log_2 B$, i.e. perform at most $4 \log_2 B$ I/Os (misalignment).
Each green tree has height between \((\log_2 B)^2\) and \(\log_2 B\), i.e. perform at most \(4 \log_2 B\) I/Os (misalignment).
Each green tree has height between \((\log_2 B)/2\) and \(\log_2 B\).

Searches visit between \(\log_B N\) and \(2\log_B N\) green trees, i.e. perform at most \(4\log_B N\) I/Os (misalignment).
Example : Recursive Layout

Cache-Oblivious String Dictionaries
Example: Recursive Layout

Cache-Oblivious String Dictionaries
Summary Cache-Oblivious Tools

<table>
<thead>
<tr>
<th>Operation</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scanning</td>
<td>$O(N/B)$</td>
</tr>
<tr>
<td>B-tree searching</td>
<td>$O(\log_B N)$</td>
</tr>
<tr>
<td>Sorting*</td>
<td>$O\left(\frac{N}{B} \log_{M/B} \frac{N}{B}\right)$</td>
</tr>
</tbody>
</table>

* requires a tall cache assumption $M \geq B^{1+\varepsilon}$

Frigo, Leiserson, Prokop, Ramachandran 1999
Brodal and Fagerberg 2002, 2003
Cache-Oblivious String Algorithms
Knuth-Morris-Pratt String Matching

Knuth, Morris, Pratt 1977

- Time $O(|T|)$
- Scans text left-to-right
- Accesses the pattern (and failure function) like a stack
Knuth-Morris-Pratt String Matching

Knuth, Morris, Pratt 1977

- **Time** $O(|T|)$
- Scans text left-to-right
- Accesses the pattern (and failure function) like a stack
- KMP is cache-oblivious and uses $O(|T|/B)$ I/Os
Edit Distance

\[
E(i, j) = \begin{cases} 
  i & \text{if } j = 0 \\
  j & \text{if } i = 0 \\
  E(i-1, j-1) & \text{if } S[i] = T[j] \\
  1 + \min\{E(i-1, j), E(i, j-1)\} & \text{if } S[i] \neq T[j]
\end{cases}
\]

- Dynamic programming
- Time \(O(|S| \cdot |T|)\)
- Space \(O(\min\{|S|, |T|\})\)
Edit Distance

$$E(i, j)$$

|   | 0 | 1 | 2 | 3 | ... | j | ... | |T|
|---|---|---|---|---|-----|---|-----|---|
| 0 | 0 | 1 | 2 | 3 | 4   | 5 | 6   | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| 1 | 1 | 0 | 1 | 2 | 3   | 4 | 5   | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 2 | 2 | 1 | 0 | 1 | 2   | 3 | 4   | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| 3 | 3 | 2 | 1 | 0 | 1   | 2 | 3   | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|    | 4 | 3 | 2 | 1 | 0   | 1 | 2   | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|    | 5 | 4 | 3 | 2 | 1   | 2 | 3   | 4 | 5 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|    | 6 | 5 | 4 | 3 | 2   | 1 | 2   | 3 | 4 | 4 | 5 | 5 | 6 | 7 | 8 | 9 | 10 |
|    | 7 | 6 | 5 | 4 | 3   | 2 | 3   | 4 | 4 | 4 | 5 | 6 | 7 | 8 | 7 | 8 | 9 |
| i | 8 | 7 | 6 | 5 | 4   | 3 | 4   | 5 | 4 | 5 | 6 | 7 | 8 | 9 | 8 | 9 | 10 |
| 9 | 8 | 7 | 6 | 5 | 4   | 5 | 4   | 5 | 6 | 5 | 6 | 7 | 8 | 9 | 8 | 9 |
| 10| 9 | 8 | 7 | 6 | 5   | 6 | 5   | 6 | 7 | 6 | 7 | 6 | 7 | 8 | 9 | 10 |
| 11| 10| 9 | 8 | 7 | 6   | 5 | 6   | 6 | 7 | 6 | 6 | 7 | 6 | 7 | 8 | 9 |
|    | 12| 11| 10| 9 | 8   | 7 | 6   | 5 | 6 | 7 | 6 | 7 | 6 | 7 | 8 | 7 | 8 |
|    | 13| 12| 11| 10| 9   | 8 | 7   | 6 | 5 | 6 | 7 | 8 | 7 | 8 | 7 | 8 |
|    | 14| 13| 12| 11| 10  | 9 | 8   | 7 | 6 | 6 | 7 | 7 | 8 | 9 | 8 | 7 |
|    | 15| 14| 13| 12| 11  | 10| 9   | 8 | 7 | 6 | 7 | 6 | 7 | 8 | 9 | 8 |
|    | 16| 15| 14| 13| 12  | 11| 10  | 9 | 8 | 7 | 8 | 7 | 8 | 7 | 8 | 9 |

Result

Cache-Oblivious String Dictionaries
## Edit Distance

| $E(i, j)$ | 0 | 1 | 2 | 3 | … | $j$ | … | $|T|$ |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| 1 | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 2 | 2 | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| 3 | 3 | 2 | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
| … | 6 | 5 | 4 | 3 | 2 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| i | 7 | 6 | 5 | 4 | 3 | 2 | 3 | 4 | 4 | 5 | 6 | 7 | 8 | 7 | 8 | 9 |

Row-by-row computation requires $O\left(\frac{|S|\cdot|T|}{B}\right)$ I/Os

Cache-Oblivious String Dictionaries
# Recursive Edit Distance

The recursive edit distance is a method for calculating the minimum number of operations required to transform one string into another. The operations include insertion, deletion, and substitution of characters. The algorithm is recursive, which means it repeatedly calls itself to solve smaller subproblems until it reaches a base case.

## Recursive Computation

The recursive computation requires $O(|S| |T|)$ I/Os, which is inefficient in terms of space. However, the algorithm can be made cache-oblivious by using a recursive table computation method. The table $E(i, j)$ represents the edit distance between substrings $S[0..i]$ and $T[0..j]$.

The base cases are:
- $E(i, 0) = i$, for $0 \leq i \leq |S|$,
- $E(0, j) = j$, for $0 \leq j \leq |T|$.

For $i > 0$ and $j > 0$, the recurrence relation is:

$$E(i, j) = \begin{cases} 
E(i-1, j-1) + 1 & \text{if } S[i] \neq T[j], \\
\min(E(i-1, j), E(i, j-1)) + 1 & \text{if } S[i] = T[j].
\end{cases}$$

The table is filled in a bottom-up manner, and the final result is in $E(|S|, |T|)$.

## Array Size

The size of the array is $|S| + |T| + 1$. This is because the table needs to handle all possible lengths of substrings from $S$ and $T$, including the case where one of the strings is empty.

## Result

The computed result is shown in the table. The result is the lower right element of the table, which represents the edit distance between the entire strings $S$ and $T$.

This method is more efficient in terms of space, making it suitable for cache-oblivious processing. The cache-oblivious string dictionaries approach ensures that the algorithm performs well on modern computer architectures with complex memory hierarchies.
### Recursive Edit Distance

The recursive edit distance algorithm is used to compute the minimum number of operations (insertions, deletions, or substitutions) needed to transform one string into another. The algorithm is implemented recursively, with the base cases being when either string is empty. The recursive formula is:

\[
E(i, j) = 
\begin{cases} 
0 & \text{if } i = 0 \\
1 & \text{if } j = 0 \\
\min(E(i-1, j), E(i, j-1), E(i-1, j-1) + 1) & \text{otherwise}
\end{cases}
\]

The computation is carried out in a table, where each cell \(E(i, j)\) represents the edit distance between the first \(i\) characters of string \(S\) and the first \(j\) characters of string \(T\). The table is filled in a bottom-up manner, with each cell depending on the values of its neighbors.

The final edit distance is found in the bottom-right cell of the table, \(E(|S|, |T|)\), where \(|S|\) and \(|T|\) are the lengths of strings \(S\) and \(T\) respectively.

#### Example

Let's consider the strings \(S = "abcab"\) and \(T = "bcab"\). We will compute the edit distance using the recursive formula.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>i</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

The final edit distance is \(E(5, 5) = 2\), indicating that the minimum number of operations needed to transform \(S\) into \(T\) is 2.
## Recursive Edit Distance

|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|
| 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 1 | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| 2 | 2 | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
| 3 | 3 | 2 | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 6 | 6 | 5 | 4 | 3 | 2 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |   |
| 7 | 7 | 6 | 5 | 4 | 3 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 7 | 8 | 9 |   |

Recursive computation requires $O(|S| + |T| + 1)$ I/Os.

---

### Cache-Oblivious String Dictionaries
## Recursive Edit Distance

The recursive edit distance is defined as the minimum number of operations required to transform a string $S$ into another string $T$, where the allowed operations are insertions, deletions, and substitutions.

The recursive formula for edit distance is given by:

$$E(i, j) = \begin{cases} 
0 & \text{if } i = 0 \\
0 & \text{if } j = 0 \\
1 + \min(E(i-1, j), E(i, j-1), E(i-1, j-1)) & \text{otherwise}
\end{cases}$$

The recursive edit distance function $E(i, j)$ is computed for all $i, j$ where $0 \leq i \leq |S|$ and $0 \leq j \leq |T|$, and the result is $E(|S|, |T|)$.

### Algorithm

1. Initialize a table $E$ of size $|S| \times |T| + 1$ with $E(i, 0) = i$ and $E(0, j) = j$ for all $i, j$.
2. For each $i$ from $1$ to $|S|$ and each $j$ from $1$ to $|T|$, compute $E(i, j)$ using the recursive formula.
3. The result is $E(|S|, |T|)$.

### Example

Let's consider the strings $S = \text{abc}$ and $T = \text{bac}$.

```
E(i, j)  0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
0   0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
1   1 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
2   2 1 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
3   3 2 1 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
    4 3 2 1 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

**Result:**

The edit distance between $S$ and $T$ is $3$.

### Complexity

The recursive computation requires $O(|S| \times |T|)$ time and space.

### Cache-Oblivious String Dictionaries

Cache-oblivious string dictionaries are data structures designed to minimize the number of cache misses during operations like insert, delete, and search. They achieve this by using a technique called auxiliary blocks, which are borrowed from the Cache-Oblivious Data Structures framework to make the data structure cache-oblivious.

### References


Recursive Edit Distance

| $E(i, j)$ | 0 | 1 | 2 | 3 | ... | $j$ | ... | $|T|$ |
|-----------|---|---|---|---|-----|-----|-----|-----|
| 0         | 0 | 1 | 2 | 3 | 4   | 5   | 6   | 7   |
| 1         | 1 | 0 | 1 | 2 | 3   | 4   | 5   | 6   |
| 2         | 2 | 1 | 0 | 1 | 2   | 3   | 4   | 5   |
| 3         | 3 | 2 | 1 | 0 | 1   | 2   | 3   | 4   |
| ...       | ... | ... | ... | ... | ... | ... | ... | ... |
| $i$       | 8 | 7 | 6 | 5 | 4   | 3   | 4   | 5   |
| 9         | 9 | 8 | 7 | 6 | 5   | 4   | 5   | 6   |
| 10        | 10| 9 | 8 | 7 | 6   | 5   | 6   | 7   |
| ...       | ... | ... | ... | ... | ... | ... | ... | ... |
| $|S|$      | 16| 15| 14| 13| 12  | 11  | 10  | 9   |

Result array of size $|S| + |T| + 1$

Cache-Oblivious String Dictionaries
Recursive Edit Distance

\[ E(i, j) \]

|   | 0  | 1  | 2  | 3  | ... | j | ... | |T| |
|---|----|----|----|----|-----|---|-----|---|
| 0 | 0  | 1  | 2  | 3  | 4   | 5 | 6   | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| 1 | 1  | 1  | 2  | 3  | 4   | 5 | 6   | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 14 | 15 |
| 2 | 2  | 1  | 1  | 2  | 3   | 4 | 5   | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 14 |
| 3 | 3  | 2  | 1  | 0  | 1   | 2 | 3   | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 |
|   | 4  | 3  | 2  | 1  | 0   | 1 | 2   | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 |

Recursive computation requires \( O(|S| + |T| + 1) \) I/Os.

Cache-Oblivious String Dictionaries
# Recursive Edit Distance

$$E(i, j)$$

|   | 0 | 1 | 2 | 3 | ... | j | ... | |T| |
|---|---|---|---|---|-----|---|-----|---|---|
| 0 | 0 | 1 | 2 | 3 | 4   | 5 | 6   | 7 | 8  |
| 1 | 1 | 0 | 1 | 2 | 3   | 4 | 5   | 6 | 7  |
| 2 | 2 | 1 | 0 | 1 | 2   | 3 | 4   | 5 | 6  |
| 3 | 3 | 2 | 1 | 0 | 1   | 2 | 3   | 4 | 5  |
|   | 4 | 3 | 2 | 1 | 0   | 1 | 2   | 3 | 4  |
|   | 5 | 4 | 3 | 2 | 1   | 0 | 1   | 2 | 3  |
|   | 6 | 5 | 4 | 3 | 2   | 1 | 0   | 1 | 2  |
|   | 7 | 6 | 5 | 4 | 3   | 2 | 1   | 0 | 1  |
|   | 8 | 7 | 6 | 5 | 4   | 3 | 2   | 1 | 0  |
|   | 9 | 8 | 7 | 6 | 5   | 4 | 3   | 2 | 1  |
|   | 10| 9 | 8 | 7 | 6   | 5 | 4   | 3 | 2  |
|   | 11| 10| 9 | 8 | 7   | 6 | 5   | 4 | 3  |
|   | 12| 11| 10| 9 | 8   | 7 | 6   | 5 | 4  |
|   | 13| 12| 11| 10| 9   | 8 | 7   | 6 | 5  |
|   | 14| 13| 12| 11| 10  | 9 | 8   | 7 | 6  |
|   | 15| 14| 13| 12| 11  | 10| 9   | 8 | 7  |
|   | 16| 15| 14| 13| 12  | 11| 10  | 9 | 8  |

Recursive computation requires $$O \left( \left\lfloor \frac{|S| \cdot |T|}{M^2} \right\rfloor \cdot \frac{M}{B} \right) = O \left( \left\lfloor \frac{|S| \cdot |T|}{M \cdot B} \right\rfloor \right)$$ I/Os

Cache-Oblivious String Dictionaries
Suffix Tree/Suffix Array Construction

Farach et al. 2000

Reduces to sorting, i.e. $\text{Sort}(N)$ I/Os
Sorting $n$ Strings of Total Length $N$

- Internal memory $O(n \log n + N)$ time

- The strings can be sorted using suffix tree construction,
  \{ acabab, aabac, bac \} $\Rightarrow$ aabac#acabab#bac$
  $i.e. cache-oblivious and $\text{Sort}(N)$ I/Os

- Cache-aware

  \[ O \left( \min \left\{ \frac{N_1}{B} \log_{M/B} \frac{N_1}{B}, K_1 \log_M K_1 \right\} + K_2 \log_M K_2 + \frac{N}{B} \right) \] I/Os

  $K_1$ short strings (length $\leq B$) with total length $N_1$

  $K_2$ long strings with total length $N_2$

Arge et al. 1997
String Dictionaries
Tries vs Blind Tries

Searches take $O(|P|)$ time in internal memory for constant sized alphabets and $O(\log n + |P|)$ time for comparison based alphabets.
The Trouble Starts...

- Tries cannot be stored cache-aware to support top-down searches in $O(\log_B N + |P|/B)$ I/Os
  Demaine et al 2004

- Can construct suffix trees cache-obliviously using $O(\text{Sort}(N))$ I/Os, but cannot search in it efficiently...

+ Cache-aware string B trees support searches in a set of strings in $O(\log_B n + |P|/B)$ I/Os
  Ferragina and Grossi 1999
String Dictionary

Queries: Search blind trie + Verify one string
Queries: Search blind trie + Verify one string
Suffix Tree

Queries: Search blind trie + Verify one suffix

Cache-Oblivious String Dictionaries
Suffix Tree

Queries: Search blind trie + Verify one suffix
Tries

Queries: Search blind trie + Verify prefix of one path
Tries

Queries: Search blind trie + Verify prefix of one path
Verifying a Prefix of a Path in a Tree
Verifying Paths in Giraffe Trees is Easy

Definition

A tree is a giraffe tree if all root-to-leaf paths share at least half of the nodes of the tree (long neck)
Verifying Paths in Giraffe Trees is Easy

**Definition**

A tree is a giraffe tree if all root-to-leaf paths share at least half of the nodes of the tree (long neck)

- A prefix of length $p$ of a path in a giraffe tree using a BFS layout can be traversed in $O(p/B)$ I/Os
Giraffe Cover of a Tree

Uses space $O(N)$ and can be constructed greedily from left-to-right using $O(N=B)$ I/Os by an Euler traversal of $T$. BFS layout of each giraffe.

A prefix of length $p$ of a path in a known giraffe can be traversed in $O(p=B)$ I/Os.
Giraffe Cover of a Tree

- Uses space \( O(N) \) and can be constructed greedily from left-to-right using \( O(N/B) \) I/Os by an Euler traversal of \( T \)
- BFS layout of each giraffe
- A prefix of length \( p \) of a path in a known giraffe can be traversed in \( O(p/B) \) I/Os
Summary so far...

String dictionary search
Suffix tree search
Trie search

\[ \text{reduce to blind trie search} \]

\[ \text{Query} : \text{Blind trie search} + O \left( 1 + \frac{|P|}{B} \right) \text{ I/Os} \]
Cache-Oblivious (Blind) Tries
Cache-Oblivious (Blind) Tries

- Partition input trie $T$ into components (generalization of heavy paths)
- $T' = \text{collapse components in } T \text{ into high degree nodes and replace by weight balanced trees}$
- Apply van Emde Boas layout out to $T'$
Cache-Oblivious (Blind) Tries

Partition input trie $T$ into components (generalization of heavy paths)

$T' = \text{collapse components in } T \text{ into high degree nodes and replace by weight balanced trees}$

Apply van Emde Boas layout out to $T'$

Search: $O(\log_B n)$ I/O
Cache-Oblivious (Blind) Tries

- Partition input trie $T$ into components (generalization of heavy paths)
- $T' = \text{collapse components in } T \text{ into high degree nodes and replace by weight balanced trees}$
- Apply van Emde Boas layout out to $T'$

**Search:** $O(\log_B n)$ I/O — ignoring searching inside components
Decomposition into Components

\[ D^0_v = \{ u \in T_v \mid \text{rank}(u) = \text{rank}(v) \land \text{depth}(u) - \text{depth}(v) < 2^{2^0} \} \]

\[ D^i_v = \{ u \in T_v \mid \text{rank}(v) - \text{rank}(u) < \varepsilon 2^i \land 2^{2^i-1} \leq \text{depth}(u) - \text{depth}(v) < 2^{2^i} \} \]
Storing and Searching Components

- Store each layer $D^i_v$ separately
- Make a giraf-decomposition of $D^i_v$
- For $D^i_v$ have a blind trie of size $O(2^{e2^i})$ (using BFS layout) to select the right giraffe-tree
- Search: $D^i_v$ search the blind trie + search in one giraffe-tree
- Distribute $D^0_v, D^1_v, D^2_v, \ldots$ in the van Emde Boas layout of $T'$
- Analysis:
  - Search in blind trie for $D^{i+1}_v$ dominated by the matched characters in $D^i_v$
  - Space in van Emde Boas layout for a subtree of size $k$ becomes $O(k^3)$
There exists a cache-oblivious trie supporting prefix queries in

\[ O(\log_B |n| + |P|/B) \] I/Os,

where \( P \) is the query string, and \( n \) is the number of leaves in the trie.

It can be constructed in \( O(\text{Sort}(N)) \) time, where \( N \) is the total number of characters in the input.

The space required is \( O(N) \).

The structure assumes \( M \geq B^{2+\delta} \).
Conclusion

- A string dictionary (trie data structure) was presented that supports queries in $O(\log_B n + |P|/B)$ I/Os. The data structure uses $O(N)$ space and can be constructed using $O(\text{Sort}(N))$ I/Os.

- Lookahead in the query string is crucial (both cache-aware and cache-oblivious)

- A giraffe cover is a simple construction allowing topdown path traversals in a tree using $O(|P|/B)$ I/Os
Open problems

- Prove a lower bound trade-off between the number of I/Os required for a query and the lookahead used

- Implementation: compare with string B-trees, tries, ternary trees, different trie layouts, ...
Thank you —Lunch!