Dynamic Succinct Tries

Rajeev Raman

University of Leicester

MatBio 2016, King’s College London
(Joint work with Andreas Poyias and Simon Puglisi)
Overview

1. Introduction

2. Succinct Data Structuring

3. Succinct Tries
Big Data vs. big data

- **Big Data**: 10s of TB+.
  - Must be processed in streaming / parallel manner.
- Data mining is often done on big data: 10s-100s of GBs.
  - Graphs with 100s of millions of nodes, protein databases 100s of millions of compounds, 100s of genomes etc.
- Often, we use Big Data techniques to process big data.
  - Parallelization is *hard* to do well [Canny, Zhao, *KDD’13*].
  - Streaming is inherently limiting.
- Instead of changing the way we *process* the data, why not change the way we *represent* the data?
Introduction

Processing big data

- Essential that data fits in main memory.
  - Complex memory access patterns: out-of-core ⇒ thrashing.
- Data accessed in a complex way is usually represented in a data structure that supports these access patterns.
  - Often data structure is MUCH LARGER than data!
  - Cannot process big data if this is the case.
- Examples:
  - Suffix Tree (text pattern search).
  - Range Tree (geometric search).
  - FP-Tree (frequent pattern matching).
  - Multi-bit Tree (similarity search).
  - DOM Tree (XML processing).
Succinct/Compressed Data Structures

Store data *in memory* in *succinct* or *compressed* format and operate directly on it.

- (Usually) no need to decompress before operating.
- Better use of memory levels close to processor, processor-memory bandwidth.
  - Usually compensates for some overhead in CPU operations.

*Programs = Algorithms + Data Structures*

- If compressed data structure implements same/similar ADT to uncompressed data structure, can reuse existing code.
Compression vs. Data Structuring

Answering queries requires an *index* in *in addition to* the data.

Space usage = “space for data” + “space for index”.

Index may be larger than the data:

- **Suffix tree**: data structure for indexing a text of \( n \) bytes.
  - Supports many indexing and search operations.
  - Careful implementation: \( 20n \) bytes of index data in worst case [Kurtz, *SPrEx '99*]

- **Range Trees**: data structures for answering 2-D orthogonal range queries on \( n \) points.
  - Good worst-case performance but \( \Theta(n \log n) \) space.
If the object \( x \) that you want to represent is drawn from a set \( S \), \( x \) must take at least \( \log_2 |S| \) bits to represent.

Example: object \( x \) is a binary tree with \( n \) nodes.
- \( x \) is from the set \( S \) of all binary trees on \( n \) nodes.
- There are \( \sim 4^n \) different binary trees on \( n \) nodes.
- Need \( \sim \log_2 4^n = 2n \) bits, or 2 bits per node.
- A normal representation: 2 pointers, or \( 2 \log_2 n \) bits, per node.

Space usage for \( x = \text{"space for data"} + \text{"space for index"}, \)
- ITLB for \( x \)
- lower-order term

and support fast operations on \( x \).
Applications of SDS in Bioinformatics

- Compressed suffix trees (many authors).
- Bowtie (short read alignment) [Langmead et al., *Genome Biol. ’09*].
- Succinct de Bruijn Graph [Bowe, Sadakane, *WABI’12*], [Boucher et al., *DCC’15*].

Commonly used as a tool for large-scale data analysis.

- Majority of the work on static data structures.
- Expensive pre-processing, fast queries.
  - Data changes $\rightarrow$ repeat pre-processing.
- Less work on dynamic data structures.
  - We focus on the “trie” abstract data type (ADT).
The “trie” ADT

- Object is a rooted tree with \( n \) nodes.
- Each node from a parent to a child is labelled with a distinct letter \( c \) from an alphabet \( \Sigma \), where \( \Sigma = \{0, \ldots, \sigma - 1\} \).
- All possible children may not be present.
- Represents a collection of strings over \( \Sigma \).

\[
\Sigma = \{0, 1, 2, 3\}, \ n = 50
\]

Operations

- \( \text{parent}(x) \);
- \( \text{child}(x, c) \);
- \( \text{desc}(x), \ \text{nextsib}(x), \ \text{prevsib}(x), \ \ldots \)
Normal Trie Representations

![Ternary search tree with nodes b, o, r, s arranged in a binary tree. Space: 4 pointers (256 bits) per node. Child: \( O(lg \sigma) \) time. ITLB = \( \lceil log_2 \left( \frac{1}{\sigma} n + 1 \left( \sigma n + 1 \right) n \right) \rceil \) \( \sim n log_2 \sigma + O(n) \) bits. \( \Delta \) One character per node.]

[Bentley/Sedgewick, SODA'97]
Normal Trie Representations

- Each node points to parent, first-child and next-sibling.
- Space: $3$ pointers ($O(\log n)$ bits/192 bits) per node.
- $child$: $O(\sigma)$ time.

---

$\sigma$ character per node.
Normal Trie Representations

- Each node has array of $\sigma$ pointers, one to each possible child.
  - Space: $\sigma + 1$ pointers per internal node.
  - \textit{child}: $O(1)$ time.
Normal Trie Representations

- **Ternary search tree** [Bentley/Sedgewick, *SODA’97*]. Siblings arranged in a binary tree.
  - Space: 4 pointers (256 bits) per node.
  - child: $O(\lg \sigma)$ time.
Normal Trie Representations

- **Ternary search tree** [Bentley/Sedgewick, *SODA’97*]. Siblings arranged in a binary tree.
  - Space: 4 pointers (256 bits) per node.
  - *child*: $O(\log \sigma)$ time.

- **ITLB** = \[\log_2 \left( \frac{1}{\sigma n+1} \frac{(\sigma n+1)}{n} \right) \] \sim n \log_2 \sigma + O(n) \text{ bits.}
  - One character per node.
Output a 1. Then visit each node in level-order and output $\sigma$ bits that indicate which labels are present. [Jacobson, FOCS’89]

Bit-string is of length $\sigma n + 1$ bits. It has $n$ 1s.

Its ITLB is $\left\lceil \log_2 \left( \frac{\sigma n + 1}{n} \right) \right\rceil \sim n \log_2 \sigma + O(n)$ bits.

Representation is static, but supports many operations in $O(1)$ time.
Dynamic Tries

- **ADT:**
  - `parent(x);`
  - `child(x, c);`
  - `add(x, c).`

- Several theoretical proposals.
  - \( n(\log_2 \sigma + 2) + o() \) bits, \( O(1) \) time operations for \( \sigma = (\log n)^{O(1)} \) [Arroyuelo et al., *Algo. ’15*].
  - \( O(n \log \sigma) \) bits and \( O(\log \log n / \log \log \log n) \) time [Jansson et al., *Algo. ’15*].
  - Also mention *wavelet trie* [Grossi/Ottaviano, *PODC’13*].

- No obvious practical solutions.
Bonsai Trees [Darragh et al., *Soft. Prac. Exp’93*]

Store trie in open hash table of $M = (1 + \epsilon)n$ entries.

- Nodes of trie reside in hash table.
- ID of a node: location where it resides.
- ID of child labelled $c$ of $x$:
  - Create key $\langle x, c \rangle$ and insert.

![Diagram of a trie](image)

```
root  a  b  c  e
  a
```

```
0 1 2 3 4 5 6 7 8 9
```

```
4,a 4,c  root 2,a 4,e
```
Succinctness?

- Hashed values are in the range \( \{0, \ldots, M\sigma - 1\} \), take \( \log_2 n + \log_2 \sigma + O(1) \) bits.
- The space usage is \( (1 + \epsilon)n(\log_2 n + \log_2 \sigma + O(1)) \) bits.
- To reduce space, use quotienting.
  - Assume no collisions in the hash function and assume the hash function \( h(x) = ax \mod M \).
  - Store \( T[i] = ax \div M \) where \( i = h(x) \).
    - \( i \) itself is the mod value.
    - Can reconstruct \( x = a^{-1}(T[i] \times M + i) \).
  - \( T[i] \) requires only \( \log_2 \sigma + O(1) \) bits!
  - Space for \( T \) is \( (1 + \epsilon)n(\log_2 \sigma + O(1)) \) bits.
- Collisions?
  - \( h(x) = h(y) \) but \( x \neq y \).
  - \( x \) may not be stored in \( T[h(x)] \).
Compact Hashing

Compact hash tables [Cleary, *IEEE Computer’83*].

- To handle collisions, use a kind of linear probing.
- Use two bit-vectors of $M$ bits each to determine $h(x)$.
  - Space for $T$ is $(1 + \epsilon)n(\log_2 \sigma + O(1))$ bits.
- Inserting takes $O(1)$ expected time but moves keys.

All descendants of a node $v$ are based upon $v$’s ID.

Need “persistent” node IDs which don’t change.

This is fixed by Darragh et al. but space becomes $(1 + \epsilon)n(\log_2 \sigma + O(\log \log n))$ bits.

- For $\sigma = 4$ and $\epsilon = 0.25$ Bonsai takes $12.5n$ bits for $n \leq 2^{64}$.
  - ITLB $\sim 3.24n$ bits.
Compact Hashing

m-Bonsai [RP, SPIRE’15]

- Collision resolution by *linear probing*.
  - Let $i = h(x)$. Try $T[i], T[i+1], T[i+2], \ldots$ until an empty location is found.
  - If $h(x) = i$ and $x$ is placed in $T[j]$ for $j > i$, store the *displacement* value $j - i$ in the $j$-th location of array $D$.
  - If $T[i]$ contains a key $x$ then $h(x) = i - D[i]$.

- On average $\sum_{i=0}^{M-1} D[i]$ is $O(M)$.
  - $D[i]$ is the average search time of a key, which is $O(1)$.
  - Maximum $D$ value can be $O(\log n)$ or $O(\log \log n)$ bits.

- Storing $D$ in an array, where each entry is maximum size will take $(1 + \epsilon)n(\log_2 \sigma + O(\log \log n))$ bits.
  - The $O(\log \log n)$ term is larger than in Darragh et al.
CD-RW Arrays

Compact Dynamic Read-Write arrays store non-negative integers. ADT:

- `create(M, k)`: returns a new array `A` of size `M` with maximum value $2^k - 1$.
- `get(i)`: returns value of `A[i]`.

Trivial $O(1)$ time `get` and `set` using $Mk$ bits. We want a space bound that is related to $S = \sum_{i=0}^{M-1} \lceil \log_2(A[i] + 1) \rceil$ bits.

<table>
<thead>
<tr>
<th>get</th>
<th>set</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(1)$</td>
<td>$O(\log v)$</td>
<td>$O(S)$</td>
</tr>
<tr>
<td>$O(1)$ exp.</td>
<td>$O(1)$ exp.</td>
<td>$O(S + M \log k)$</td>
</tr>
<tr>
<td>$O(\log v)$ exp.</td>
<td>$O(1)$ exp.</td>
<td>$O(S)$</td>
</tr>
</tbody>
</table>
CD-RW Arrays and Bonsai

New solutions are based on compact hashing and appear to work well in practice. E.g. for $\sigma = 5$:

- ITLB $\sim 3.61n$ bits.
- Bonsai $\sim 12.5n$ bits.
- m-Bonsai $\sim 6.7n$ bits.
- TST $\sim 264n$ bits.

Bonsai and m-Bonsai are as fast as TST for insert and navigation!
m-Bonsai, dynamic?

- Bonsai uses an array of size $(1 + \epsilon)n$, where $n$ is the number of nodes in the trie.
  - Create a new array of size “double” the size of the old array.
  - Copy trie from old array to new array.
- To copy the trie, you have to be able to traverse it.
  - No nextsib operation in (m-)Bonsai.
  - Traversal takes $O(n\sigma)$ time.
- No solution mentioned even in the original paper!
- New solution:
  - A key $x$ in the hash table is a pair $(i, c)$ where $i$ is the index of the parent.
  - Sort all pairs using radix sort; this puts all the labels of the children of a node together.
  - Copying in $O(n)$ time.
Conclusion

- Presented an improved version of a compact dynamic trie.
  - CD-RW array is an interesting sub-problem.
- New version is about half the memory of the current best (order of magnitude less than naive) and as fast as both predecessors.
- Open questions:
  - Get closer to ITLB in practice.
  - Does “Patricia” trie make any sense here?
  - Need strong assumptions about hash tables.
    - These assumptions provably untrue in worst case in Bonsai.
    - New model of analyzing hash functions needed.
    - There are probably good hash functions, but can they support quotienting?