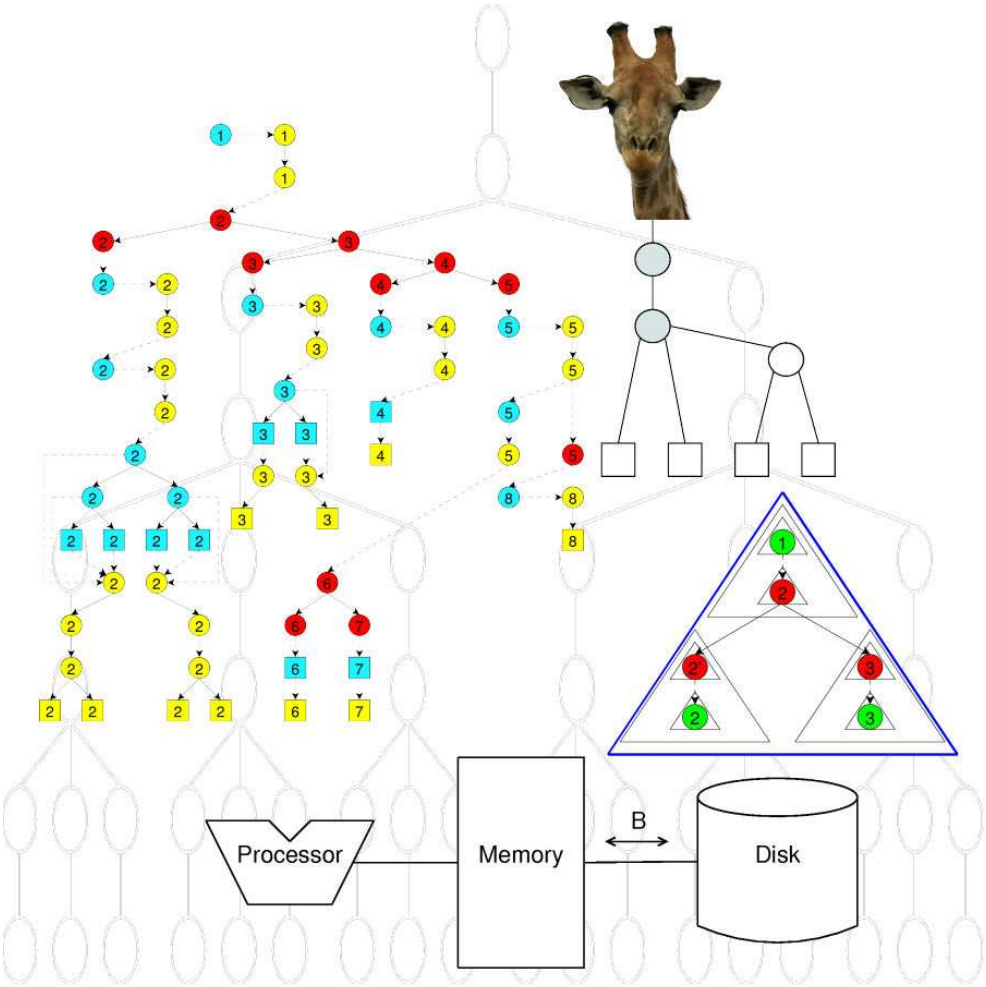
Department of Computer Science
Faculty of Science
University of Aarhus

Cache Oblivious String Dictionaries

Master Thesis



written by
Bo S. Carstensen (20002687)
Casper Torndahl (20012057)

supervised by
Gerth S. Brodal

February 20, 2007

This page intentionally left blank[1]

[1] http://www.this-page-intentionally-left-blank.org

Contents

List of Figures

List of Tables

Abstract

This thesis presents an implementation and experimental study of the static cache oblivious string dictionary found in [Brodal and Fagerberg, 2006]. Theoretically a root to leaf search path in the cache oblivious string dictionary is performed in $O(\log_B(n) + |P|/B)$ I/Os, where B is the block size, n the number of strings in the dictionary and P the query string. This bound is tested by a variety of experiments using the cache oblivious string dictionary structure and a naive trie structure. The implementation cover the cache oblivious layout and search algorithm.

Given a trie as input a cache oblivious string dictionary is constructed using the data structures blind tries, giraffe trees, weight balanced trees (Huffman trees) and weight balanced search trees (Leaf oriented optimal binary search trees). The structure is laid out using a van Emde Boas layout. The I/O bound is archived using redundancy, i.e paths in the trie is stored multiple times. Even so the cache oblivious string dictionary structure uses only linear space.

The cache oblivious layout in this thesis is not build cache oblivious. Therefore, the Huffman tree is used instead of the tree from [Brodal and Fagerberg, 2006] Section 5.

The experiments show execution times for various parameters for the cache oblivious layout, in a attempt to establish the best. The result of the experiments show that a cache oblivious layout have superior execution time compared to a naive implementation of a trie.

Introduction

People often think of a heavy book when hearing the word dictionary. For people in computer science a dictionary is equivalent with the trie data structure. The trie structure stores strings and supports queries for these. The content of the strings are not limited to a certain type. They can for instance contain DNA sequences, integers or simply words from a Shakespeare play. The challenge is to construct the trie, so that the searches for a prefix query is efficient. Especially when the trie is laid out in external memory. The best known bound for a prefix query is $O(|P| + \log(n))$ for unbounded alphabets in internal memory.

Over the last decade the interest for I/O efficient algorithms has increased. This is mainly due to the increasing amount of data needed to be processed in still shorter time. Even though the capacity of caches and memory layers keep increasing, efficient queries on stored data is still an issue. Not only in external memory but also in main memory. Cache oblivious algorithms are attempts to store data allowing queries to be answered efficiently, both in memory but especially in external memory.

It can be proved that it is not possible to lay out a trie in external memory achieving a query time of $O(\log_B(n) + |P|/B)$ I/Os in the worst case. It is possible though, by other means than the trie structure. Using structures like blind tries, giraffe trees and weight balanced search trees, [Brodal and Fagerberg, 2006] achieved a query time of $O(\log_B(n) + |P|/B)$ I/Os worst case. This thesis is an implementation of the theory in [Brodal and Fagerberg, 2006]

The thesis is composed in five parts. The first part describes the data structures and models used in the construction of the cache oblivious string dictionary. The next part concerns the construction of the cache oblivious string dictionary together with proof of the time bound $O(\log_B(n) + |P|/B)$ and space bound $O(N)$. The subject of the third part is the implementation of the cache oblivious string dictionary together with the search algorithm. The next to last part presents the results of the experiments, where different values of parameters have been tested. The final part is the appendix. In the appendix a user manual is included together with tables showing the output from construction of the various layouts.

Part I

Structures and models

1 Analysis Models

To analysis an algorithm, an analysis model must be described. Three of the most widely used are the *von Neumann RAM model*, *the I/O model* and the *cache oblivious model*.

1.1 Von Neumann RAM Model

The Von Neumann RAM[2] model, [von Neumann, 1945], is used to analyse operations done in main memory. It is assumed that only one processor is used and no concurrent operations is allowed. Each instruction is charged a cost of units, making it possible to analyse the cost of algorithms working in main memory.

Before the RAM model can be used to analyse the running time of an algorithm, the instruction set must be described and the cost of each instruction specified. Basic instructions as arithmetic (`add`, `subtract`, `multiply`, `divide`, `remainder`, `floor`, `ceiling`,), movement (`move`, `copy`, `store`) and control (`if`, `if else`, `return`) are typically charged a constant number of units. The cost of instructions like `sort` is depended on the number and times the basic operations are used and the cost of these.

The closer to reality the instructions set with the cost of each instruction is, the more realistic will the analysis in the RAM model be.

1.2 I/O Model

Computer storage is typically ordered in a hierarchy, where each layer acts as a cache for the next larger but also slower layer. The I/O model is used for modelling data transfers between these layers of storage. This cannot be done by the RAM model, since it only concerns operations done in one layer.

The most commonly used I/O model is the two layered model [Aggarwal and Jeffrey, 1988], where the first layer is fast and of size M and the second layer is slow but infinitely large. Data is transferred between the layers in blocks of B elements. It is only possibly to do computation on the elements in the first layer. Figure 1 shows this model.

In this model operations done on elements in the first layer is free of charge while data transfers, i.e. reading or writing, is charged by a cost. The two-layered model is widely used since the analysis in this model easily extends to models containing more layers.

[2]Random Access Machine.

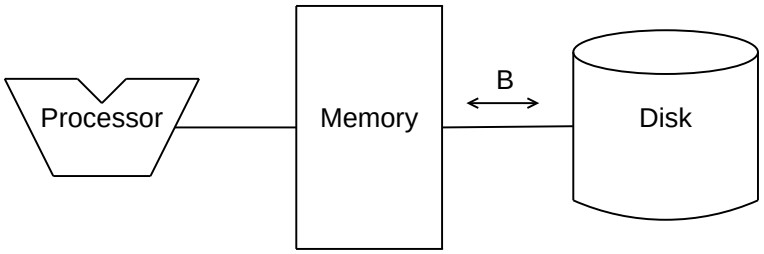


Figure 1: The I/O Model

1.3 Cache-Oblivious Model

The cache oblivious model [Frigo *et al.*, 1999] is a generalisation of the I/O model. Algorithms using the cache oblivious model are not allowed to assume anything about the values of B and M. This means these algorithms must be described in the RAM model, but analysed in the I/O model.

Since nothing is known about M, cache replacements are assumed to happen automatically by an optimal offline cache strategy. The beauty of the cache oblivious model is that since the analysis applies for any M and B, it applies for all layers of memory.

1.4 Cache misses

Most modern day CPUs have several layers of cache. When a CPU encounter data it needs it will first search the first layer of cache. If the data is not present there, it will proceed to the next layer and so forth. If the data cannot be found in a layer a cache miss occur, indicating that the data needs to be fetched.

When a block of data is loaded into a cache layer, it replaces another a block of data. Which block to choose is determined by an eviction strategy. One of these strategies is the *LRU*[3] which chooses the data block least recently used. Another strategy is the *FIFO*[4] which chooses the block who has been in the cache longest.

1.5 Data prefetching

As most CPUs are build around a pipeline architecture a cache miss can slow down the process significantly. Therefore most CPUs have prefetching mechanism [Pan *et al.*, 2007]. This mechanism tries to predict the data

[3]Least Recently Used.
[4]First In First Out.

needed next and then loading it in advance. The data is loaded into a special buffer in the level 2 cache. If a cache miss occurs this buffer is checked before the cache layer. Algorithms considering the prefetching mechanism can sometimes outperform cache oblivious and cache aware algorithms.

2 Trie

The *Trie* structure[5] [Fredkin, 1960] is a tree structure used for storing a set of strings. After storing the strings it is possible to search in the tree, making it ideal as a dictionary. Figure 2 shows an example of a trie.

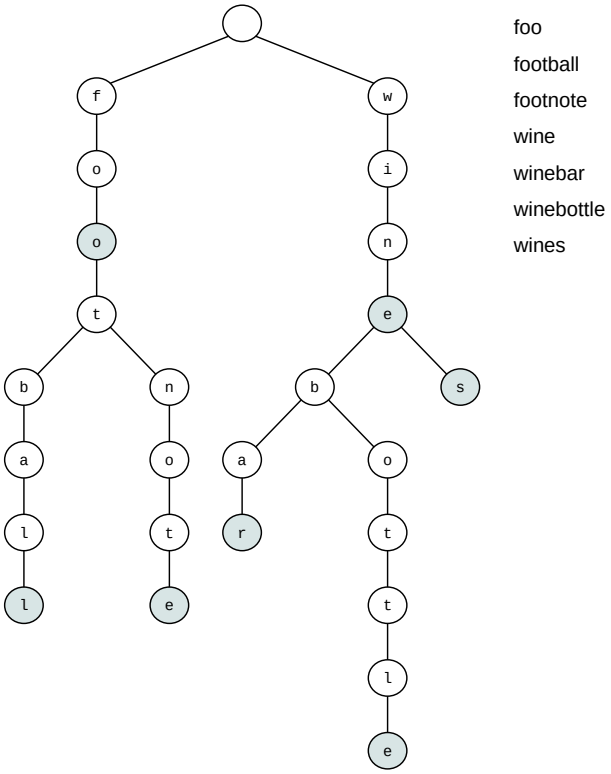


Figure 2: A trie example showing the tree where the strings "foo", "football", "footnote", "wine", "winebar", "winebottle", "wines" are inserted. A shaded node indicates that a string has ended at this node, i.e. the node is marked.

The number of children[6] for each node is determined by the size of the alphabet used. Each node can have as many children as there are characters in the alphabet[7].

2.1 Search

A search for a string is done by traversing the trie top-down while scanning the string left to right. If the current node visited has a child with the

[5] Also known as a *Radix tree*.

[6] The children of a node are the nodes right beneath it.

[7] If the alphabet is infinite, a node can arbitrary many children.

scanned character, the child is visited and the next character is scanned. If no such a child exists, the string is not in the trie. When the end of the string is reached, the current node is examined to check if it is marked or not. If it is marked, the trie contains the string.

2.2 Insert

When a string is inserted into the trie, the trie is traversed top-down while scanning the string left to right. If the current node has a child containing the scanned character, this child is visited and the next character is scanned. If no such child exists, a new child is created containing the scanned character. The new child is then visited and the next character is scanned. If there are no more characters in the string the child is marked, indicating that a string has ended at this node.

2.3 Complexity

Let $S = \{s_1, s_2, \ldots, s_m\}$ be the set of inserted strings, where the length of s_i is $|s_i|$. In the worst case, the space usage is

$$\sum_{i=1}^{m} |s_i| \in O(|S|)$$

since all strings can start with a unique character, making it impossible to share any of the nodes in the trie.

Since all characters must be examined, inserting a string s_i takes time linear to the length of the string $|s_i|$ together with the time taken to search among the children at each node. If each node contains a balanced search tree of the children[8], searching takes $\log(n)$, where n is the number of children. The total time used when inserting a string s_i is

$$O(|s_i| \cdot \log(n))$$

The time used to search for a string is identically.

If the alphabet is finite, the children can be stored using a hash table[9]. Searching among the children is thereby done in constant time. Therefore, the time used to insert or search for a string s_i is $O(|s_i|)$.

[8] Assuming some order of the alphabet, making it possible to compare characters lexicographically.

[9] Or something similar for instance a vector.

3 Blind Trie

The *Blind Trie* structure[10] [Morrison, 1968] is used for storing strings in a tree structure. It is usually constructed from a trie by eliminating all nodes with only one child, i.e. collapsing nodes.

As with the trie, it is possible to search in the blind trie. In this section two different structures are described. The first is the standard version of a blind trie while the second is slightly altered to satisfy the results in [Brodal and Fagerberg, 2006].

3.1 Standard version

In the standard version, the characters from the collapsed nodes are stored on the edges. The leaves contains the remaining characters of the inserted string, if any. The root node are always an ε-node. Figure 3 shows the constructed blind trie, when given the trie from figure 2 as input.

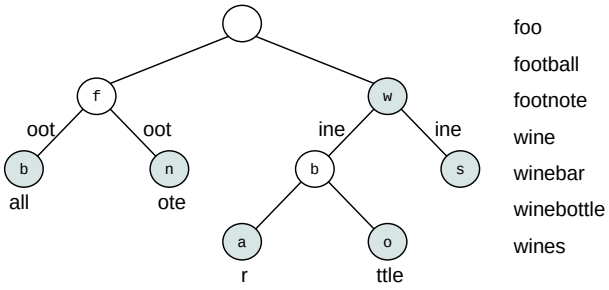


Figure 3: A blind trie example showing the (collapsed) tree where the strings "foo", "football", "footnote", "wine", "winebar", "winebottle", "wines" are inserted. The characters from the collapsed nodes are stored on the edges. Leaves stores the rest of the characters, if any. A shaded node indicates that a string has ended at this node, i.e. the node is marked.

3.1.1 Searching

A search in the blind trie structure is similar to a search in the trie structure, since all the information from the original trie is present. The blind trie is searched top down while the string is scanned left to right. When taking a path from one node to another, the characters on the edge has to be checked against the corresponding characters in the string.

[10] Also known as a *Patricia trie*.

When the end of the string is reached, two cases exists. Either the search ends in a node which has to be examined to see if it has been marked. If the node is a leaf the characters contained in the leaf are checked against the characters in the string to see if the string matches. It could also be the case that the last character in the string is on the traversed edge, and a way of checking whether or not characters on the edge has been marked is needed.

3.1.2 Inserting

Inserting a string in a blind trie is possible, as all string information is contained within the structure. It requires splitting edges, insertion of at most two new nodes and a few updates local to the inserted node.

3.1.3 Complexity

Since a standard blind trie stores the same number of characters as the trie given as input, it uses the same amount of space as the trie. Searching and insertion bounds are the same as the trie, as it might not be possible to collapse any nodes of the input trie. Replacing an edge can be done in constant time.

3.2 Altered version

Another way of constructing a blind trie, is to store the number of collapsed nodes on the edges. Figure 4 show an example of this, also given the trie from figure 2 as input. The leaves only contain one character. The rest are omitted, if any.

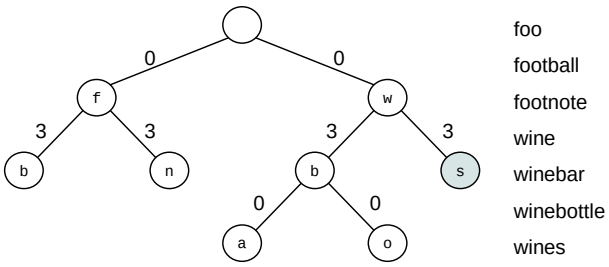


Figure 4: A blind trie example showing the (collapsed) tree where the strings "foo", "football", "footnote", "wine", "winebar", "winebottle", "wines" are inserted. The number of collapsed nodes is stored on the edges. A shaded node indicates that a string has ended at this node, i.e. the node is marked.

3.2.1 Searching

The altered version structure, where only numbers of collapsed nodes are stored, makes searching somewhat incomplete. It is only possible to partly check whether a string is present or not, since only some of the characters in the string can be verified. In order to do a complete check of the string, another structure is needed to fill in the blanks.

The check can be done in the same way as with the standard blind trie. The blind trie is searched top-down while scanning the string left to right, only checking the characters in the nodes with the corresponding characters in the string.

3.2.2 Inserting

Inserting in the altered version is not possible without an additional data structure, since the characters between nodes are missing, making it impossible to replace edges.

3.2.3 Complexity

The altered version uses in the worst case the same amount of space as a trie, as it might not be possible to collapse any nodes of the input trie. Using this argument again, searching is also the same as the trie.

4 Giraffe Tree

A *Giraffe Tree* [Brodal and Fagerberg, 2006] is defined as a tree having at least half of its nodes as ancestors to all the leaves. An example of this is shown in Figure 5.

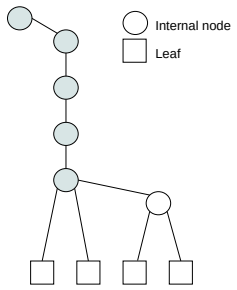


Figure 5: An example of a giraffe tree. The shaded nodes are that half of the nodes which must be ancestors of all the leaves.

Giraffe trees are used together with the altered version of blind tries. Since the blind trie does not store the characters of the collapsed nodes, it is only possible to partly check whether a string is contained in the blind trie or not. Using a set of giraffe trees covering the input trie completely, it is possible to check a string as the giraffe trees contains all the characters from the input trie. Each node in a blind trie refers to a corresponding giraffe tree, i.e. a giraffe tree covering the same node. The idea is to make a fast check of the string at certain positions in the blind trie, and if this check is successful do a thorough check of the string in a giraffe tree.

4.1 Covering a trie with giraffe trees

Covering a trie by a set of giraffe trees can be done in different ways. Figure 6 show an algorithm doing this in a greedy manner. $T^{i,j}$ is denoting a tree covering the leaves from i to j.

The algorithm scans the leaves left to right maintaining a set of leaves covered by a giraffe tree. In each iteration it is checked whether the set including the next leaf is still covered by a giraffe tree. If the set is still covered by a giraffe tree, the leaf is added to the set. If not, the giraffe tree is outputted, the set emptied, and the leaf added to the set.

Each leaf in a trie is covered by exactly one giraffe tree, while the internal nodes can be covered by more than one. Figure 7 shows the covering of a trie using the greedy algorithm.

```
i = 1
while(i <= n) do{
   j = i
   while(j < n and T^{i;j+1} is a giraffe tree) do{
      j = j+1
   }
   output T^{i;j}
   i = j+1
}
```

Figure 6: Algorithm for covering a tree with giraffe trees.

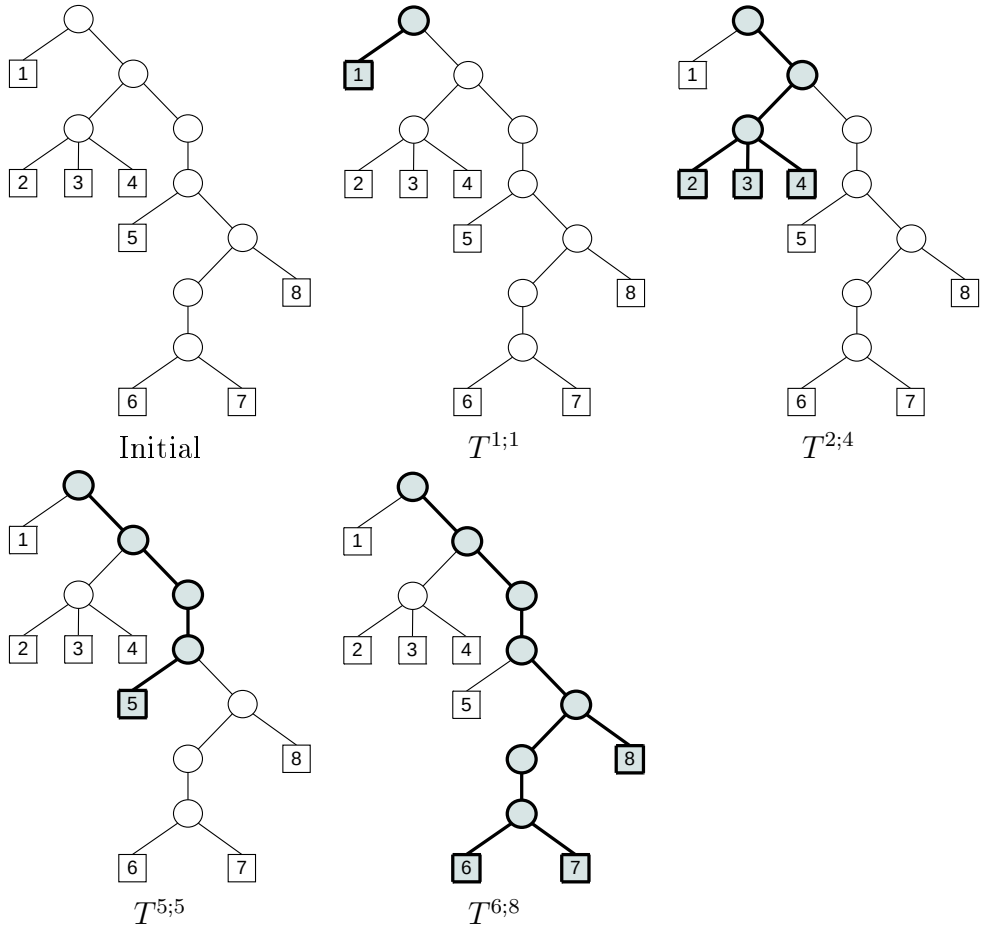


Figure 7: Example of tree covering by giraffe trees.

4.2 Searching

When validating a string, the blind trie is first traversed from root to leaf, partly checking the string against the characters stored in the nodes of the blind trie. If this check is successful, the giraffe tree attached to the blind trie leaf is traversed from root to leaf, validating the string completely. As the giraffe tree is just a search tree, the traversal is done like any other.

The advantage of using the giraffe tree, is that when validating a string in a giraffe tree, half of the string is stored in the neck of the giraffe tree. As the nodes in the neck are unary, validating the nodes in the neck is a matter of scanning.

5 Weight balanced trees

A weight balanced tree is a tree balanced by the weights of its nodes. It is not a specific data structure, but a term used to describe that the nodes are placed accordingly to a certain weight. A 'weight' of a node could be how often it is searched for in a search tree. Therefore, it would be an advantage to place 'heavy' nodes, so that they are found early in a search through the tree.

The goal of constructing a weight balanced tree is to minimise the total weight of the tree. The total weight, W, for a tree with n nodes, $m_1, m_2, \ldots, m_n$, inserted is defined as

$$W = \sum_{i=1}^{n} |d_i| \cdot w_i$$

where $|d_i|$ is the depth of node m_i having weight w_i. As a weight balanced tree is balanced by weights and not height, it is rarely the case that it has logarithmic height, i.e. $\log(n)$.

Some weight balanced tree are also search trees. If the order of the nodes are taken into consideration when constructing the tree, it can be possible to search in the tree afterwards. Constructing a search tree often results in longer construction time or a higher total weight.

5.1 Huffman tree

A Huffman tree, [Huffman, 1952] is a weight balanced tree and is introduced in an algorithm for creating an optimal prefix code, known as the Huffman code. The algorithm encodes the prefixes using a binary tree, placing the prefixes that occur the most at the top of the tree. This tree is known as the Huffman tree. The code for the prefix is then the binary representation of the path down the tree. The Huffman code is often used in data compression.

5.1.1 Constructing a Huffman tree

A Huffman tree is constructed by first inserting the weighted nodes into a priority queue, and then repeatedly merging two nodes. The nodes are merged by making the two lightest nodes in the priority queue children of a new node, which has the sum of its children weights as its weight. This new node is then inserted into the priority queue, and the merging continues. When all nodes are merged into one tree the merging stops. The algorithm in Figure 8 creates a Huffman tree in this greedy manner.

```
Insert all weighted nodes into priority queue Q

while(1 < Q.size()) do{
  node left = Q.min()
  node right = Q.min()
  Q.insert(new node(left, right, left.weight + right.weight))
}

return Q.min()
```

Figure 8: Algorithm for the Huffman tree.

An example of a Huffman tree is shown in Figure 9. Each node has a key associated showing that searching efficiently in a Huffman tree is not always possible, as the keys are not sorted left to right in the final tree.

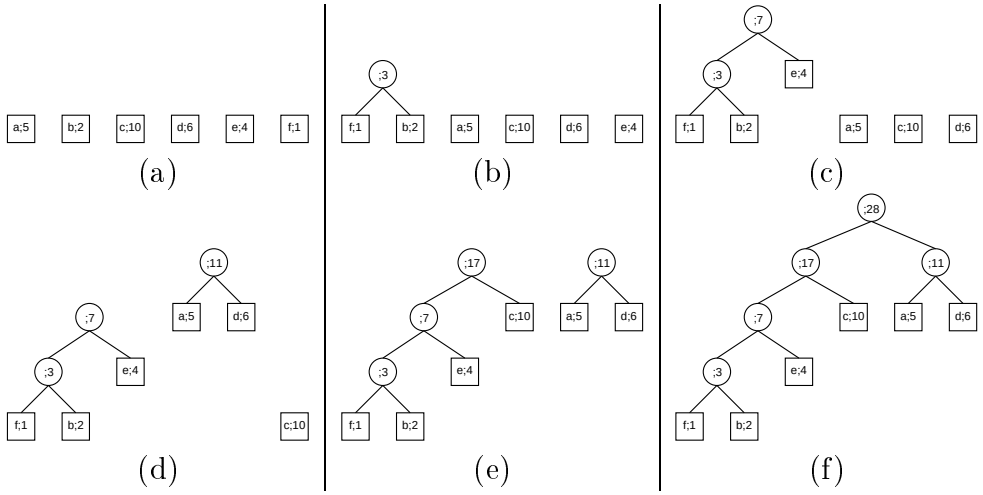


Figure 9: Example of constructing a Huffman tree.

5.1.2 Complexity

Constructing a Huffman tree takes time $O(n \log(n))$, where n is the number of leafs in the tree, as the priority queue needs to sort all the nodes. The merging takes time $O(n)$ since only $n-1$ merges are needed to construct the tree.

5.2 Article tree

The article tree is taken from the algorithm in Lemma 5.1 from [Brodal and Fagerberg, 2006] and is a weight balanced search tree. It is constructed in time $O(n)$ and the depth of a leaf with key k_i is $2 + 2\lceil \log(W/w_i) \rceil$, where w_i is the weight of k_i and

$$W = \sum_{i=1}^{n} w_i$$

for n leaves. See Section 11 for a detailed analysis.

Figure 11 shows the algorithm constructing the article tree. The leaves in the tree contain the original values, while the internal nodes are used for directing a search down to the right leaf. The `rank`-function used in the algorithm is defined as

$$\text{rank}(w) = \lceil \log(w) \rceil$$

The algorithm takes a list of sorted keys as input. It iterates through the list while maintaining a stack of trees where the ranks of the trees are strictly decreasing from bottom to top. For each key the algorithm determines whether zero or more linkings should take place. This is done by examining the rank of the current key and the rank of the tree on top of the stack. A linking links the two trees at the top of the stack into one and pushes the new tree onto the stack.

Figure 10 shows an example of the algorithm. The keys and weights are the same as used in the Huffman example.

key	a	b	c	d	e	f
weights	3	1	4	3	2	1

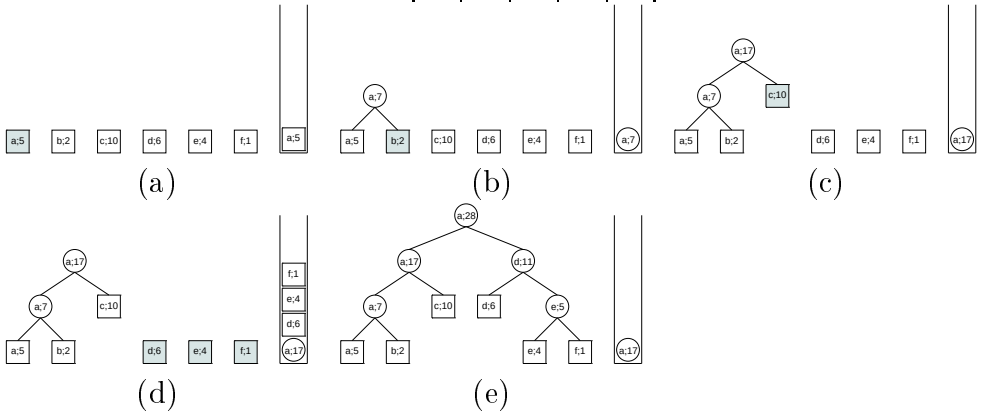


Figure 10: Example of constructing a weight balanced search tree.

```
function link(SearchTree t1, SearchTree t2){
  weight = t1.weight + t2.weight
  key = largestKeyInTree(t1)

  st = new SearchTree(key, weight)
  st.left = t1
  st.right = t2

  return st;
}

L = list of pairs (key, weight) sorted by keyvalue
S = empty stack of search trees

foreach (k,w) in L do{
  if(S.empty() or rank(w) < rank(S.top().w)) do{
    S.push(new SearchTree(k, w))
  }
  else{
    st = lowest tree in S for which rank(st) <= rank(w)

    if(st != S.top()){
      while(st is not involved in a link) do{
        // Link two top trees
        S.push(link(S.pop(), S.pop()))
      }
    }

    if(rank(S.top().w) <= rank(w)){
      S.push(new SearchTree(k, w))
      S.push(link(S.pop(), S.pop()));

      while(two top trees in S are of same rank and 1 < S.size()) do{
        S.push(link(S.pop(), S.pop()));
      }
    }
    else{ // rank(w) < rank(S.top().w)
      while(two top trees in S are of same rank and 1 < S.size()) do{
        S.push(link(S.pop(), S.pop()));
      }

      S.push(new SearchTree(k, w))
    }
  }
}

while(1 < S.size()) do{
  S.push(link(S.pop(), S.pop()));
}
```

Figure 11: An algorithm for constructing a weight balances search tree.

5.3 Optimal binary search tree

A optimal binary search tree is a tree whose expected search cost is the smallest. Given a set of keys with different probabilities, it is constructed by exhaustively checking all possible trees using dynamic programming[11]. The tree with the smallest expected search cost is not necessary the tree with the smallest overall height or has root the key with the highest possibility.

The foundation for the optimal binary search tree is the observation, that the two subtrees of an optimal binary tree also are optimal. Given a set of sorted keys $k_1, k_2, \ldots, k_n$ where $k_i < k_{i+1}$, one of these must be at the root. If this is k_i then the keys smaller than k_i must form an optimal binary search tree, and the same with the keys bigger than k_i. Figure 12 illustrates this idea. In the figure, key i have been chosen to be the root, in the left subtree key j and in the right key l.

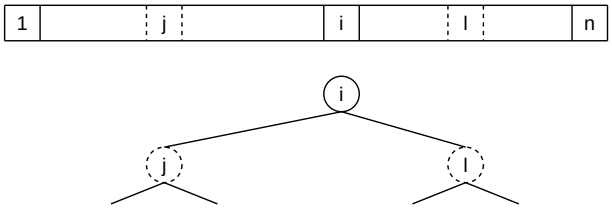


Figure 12: The basic idea behind the construction of a optimal binary search tree.

To find the optimal tree, all keys are tested. To avoid this from taking too long as many subtrees are the same for different roots, every time a subtree is found to be optimal its result is stored in a table. The size of this table is $O(n^2)$. The next time the subtree has to be calculated, the result can be found in the table avoiding the calculation.

For instance when examine key k_5 as the root of the tree containing the keys $k_1, k_2, \ldots k_m, 5 \leq m$ the optimal root for subtree $k_1 \ldots k_4$ has to be found. Later when examine another key $k_i, 5 < i$ as root, the optimal subtree for key $k_1, k_2, \ldots, k_4$ can be found by a lookup in the table storing all calculated subtrees.

The algorithm uses not only the set of keys $k_1, k_2, \ldots, k_n$ but also a set of dummy keys $d_1, d_2, \ldots, d_{n+1}$. The dummy keys are placed at the bottom of the tree as leaves. A leaf represents a search not in the tree, i.e. reaching

[11] The algorithm is not presented in this thesis. It can be found in [Cormen *et al.*, 2003] page 361.

a dummy node in a search means, that the search is unsuccessful. There are $n+1$ dummy keys as all failed searches must be directed to a dummy key. The order of the keys are

$$d_1 < k_1 < d_2 < k_2 < d_3 < \ldots < d_n < k_n < d_{n+1}$$

The algorithm uses the probabilities of each key and dummy key to determine the expected search cost.

The time used to construct and find the optimal tree is $O(n^3)$ when using dynamic programming and storing of previous results. This comes from three nested `for`-loops and only $O(1)$ lookups in the table.

5.4 Leaf oriented optimal binary search tree

A leaf oriented optimal binary search tree is constructed in the same way as an optimal binary search tree. The only difference is that the probabilities of the keys $k_1, k_2, \ldots, k_n$ are stored in dummy keys $d_1, d_2, \ldots, d_n$, i.e. $E(d_i) = E(k_i)$. Afterwards the tree is constructed using the $d - i$ dummy keys and the $i - 1$ first keys. These keys are all given the probability 0. The construction algorithm is the same as in Section 5.3.

6 van Emde Boas Tree Layout

The van Emde Boas layout [van Emde Boas *et al.*, 1977] is a recursive method of doing a layout of a binary tree. The layout is often used in cache-oblivious algorithms, since the layout is well suited for these algorithms.

The van Emde Boas layout of a binary tree is done recursively by first halving the tree T into a top T_1 and bottom trees $T_2, T_3, \ldots, T_n$. Then the top T_1 is recursive laid out followed by a recursive layout of the bottom trees $T_2, T_3, \ldots, T_n$ in a left to right order, Figure 13. The recursion stops when the height of the tree drops below a certain threshold, for instance when the height is 1.

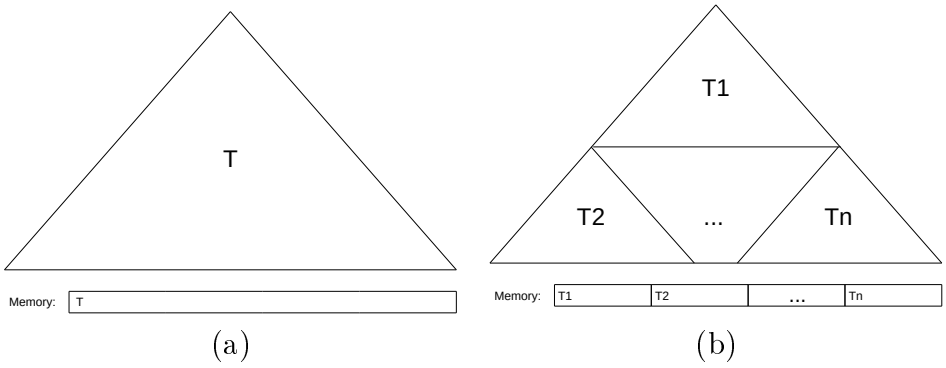


Figure 13: The theoretically van Emde Boas layout.

The tree is laid out in memory the same way it is recursively traversed. This is indicated by the `Memory` bar at the bottom of Figure 13 (a) and (b). The order of the trees is important. Following a search path root to leaf in the tree can be done by scanning forward in memory, i.e. it is never necessary to search backward.

To clarify this, Figure 14 shown an example of a binary tree layout. The tree in (a) is first halved into top and bottom resulting in the five trees in (b). These are further divided into top and bottom, and then laid out in memory, as the trees now have height one (c)[12]. Now searching the path `acjl` is a matter of scanning or following pointers forwards (d).

[12] Only the top tree T_1 is showed.

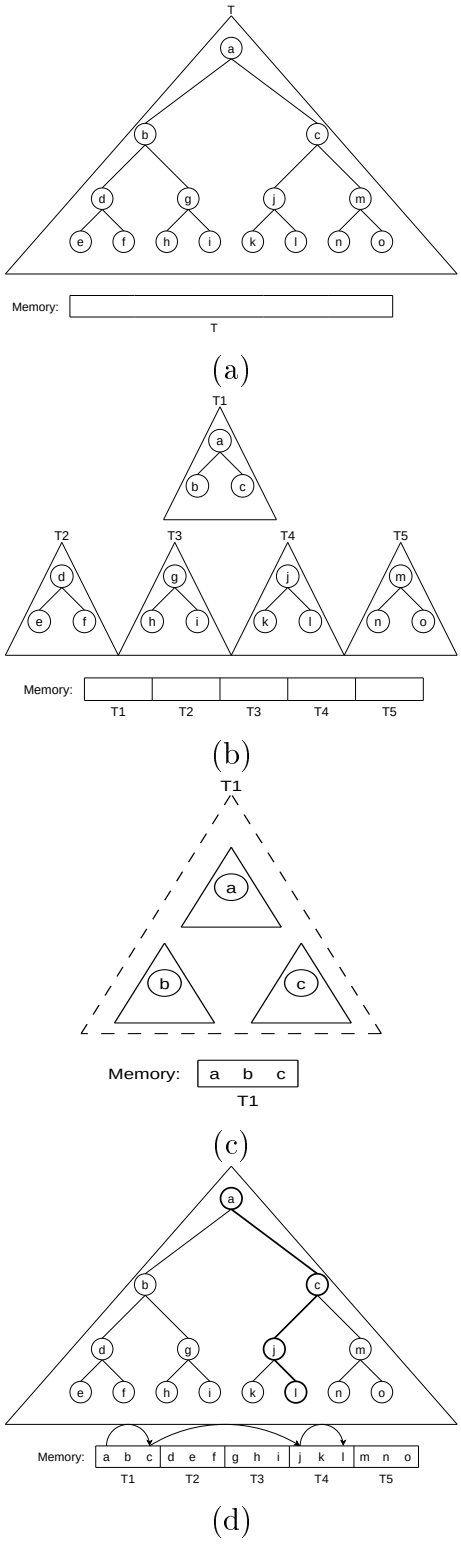


Figure 14: An example on a van Emde Boas layout.

Part II

Cache-Oblicious String Dictonary

7 Previous work

The trie structure provides a string dictionary structure which can be used in pattern matching such as prefix searches. The time cost of constructing and searching in the RAM model has been known for some time, but is still not settled in the I/O and cache oblivious model.

In the RAM model, the search time for a string P in a string dictionary structure over n strings is $O(\log(n) + |P|)$ for unbounded alphabets, and $O(|P|)$ for bounded. The corresponding construction time is $O(n \log(n)+|N|)$ for unbounded and $O(N)$ for bounded alphabets, where N is the total length of all the inserted strings. This is achieved using weight balanced search trees to store the children at each internal node and the telescope property. A search path P from root to leaf cost

$$\sum_{i=1}^{|P|} \left(1 + \log\left(\frac{w_i}{w_{i+1}}\right)\right) = |P| + \log\left(\frac{w_1}{w_{|P|}}\right) \leq |P| + \log(n)$$

where w_i are the total weight of the weight balanced search trees storing the children at node i.

A suffix tree can be constructed[13] in $O(\text{sort}(N))$[14] in both the I/O and cache oblivious model. However, searching is not trivial. It can be proved that it is not possible to lay out a trie in external memory achieving a worst case search time of $O(\log_B(n) + |P|/B)$. Using the string B-tree [Ferragina and Grossi, 1999], which is a combination of a B-tree and a blind trie, it can be achieved in the I/O model. The B-tree depends heavily on the value of B, making it useless in the cache oblivious model.

[13] Using sorting and scanning steps, [Farach-Colton *et al.*, 2000].

[14] Or more precise $O(N/B \log_{M/B}(N/B))$.

8 Overview of structure

In this section an overview of the structure is given. Starting with a trie structure T, the structure is decomposed into connected components. Each component contains blind tries and giraffe trees. To connect the components weight balanced search trees are used. Figure 15 shows an example of a trie. The trie will be used as an example in the rest of the thesis.

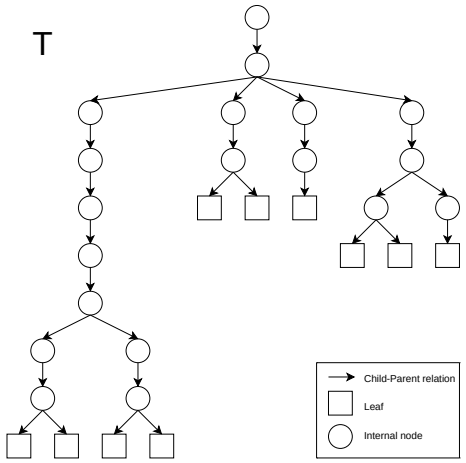


Figure 15: The input trie T with the labels omitted for simplicity.

8.1 Definitions

Before describing how to decompose an arbitrary rooted tree into components and layers a few definitions is needed.

Let T be a tree[15], v a node in T and T_v the subtree rooted at v. Then n_v is the number of leaves in the subtree T_v. The depth of v, depth(v), is the number of edges on the path from v to the root. The rank of v, rank(v), is defined as

$$\text{rank}(v) = \begin{cases} 0 & \text{if } n_v = 0 \\ \lceil \log(n_v) \rceil & \text{else} \end{cases}$$

Figure 16 shows n_v and rank(n_v) of the example trie from Figure 15.

8.2 Partition a tree into components

The components of a tree are identified recursively top-down, starting with the root of the tree. From the root the nodes are divided into *strata* and *can-*

[15]For instance a trie.

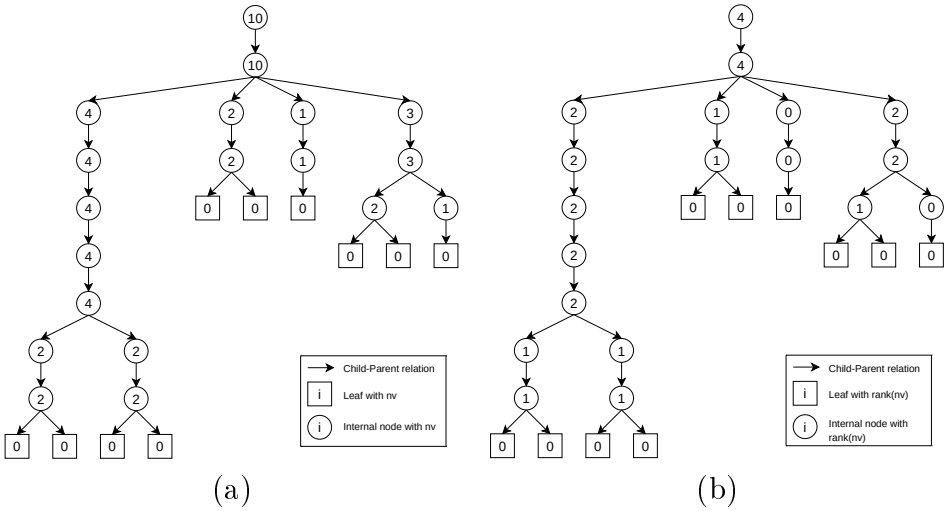


Figure 16: (a) The example trie with n_v for each node and (b) the rank of each node.

didates for the component are identified. The connected candidates starting at the root form the component. Non-candidate nodes having their parent inside the component form new roots in new components. For each new root new strata and candidates are found.

Let r be the root of the tree, or a non-candidate node, whose parent is assigned to a component. Starting at r the nodes in T_r is divided into strata by a depth condition. Let the node in question be denoted v. If

$$\text{depth}(v) - \text{depth}(r) < 2^{2^0}$$

then node v belong to strata 0. If this is not the case, then v belongs to strata i for which the following is true

$$2^{2^{i-1}} \leq \text{depth}(v) - \text{depth}(r) < 2^{2^i}$$

for $i = 1, 2, \ldots$. Figure 17 shows the depths and strata for the nodes in the example trie, where r is the root.

When the nodes in T_r are divided into strata it is possible to find the candidates for the component. Let $\varepsilon \in (0, 1]$ be a constant, used to influence the size of components. For small values of ε the components will contain few nodes, and for large values more nodes. A node in T_r is a candidate, if the following is true

$$\text{rank}(r) - \text{rank}(v) < \varepsilon 2^i$$

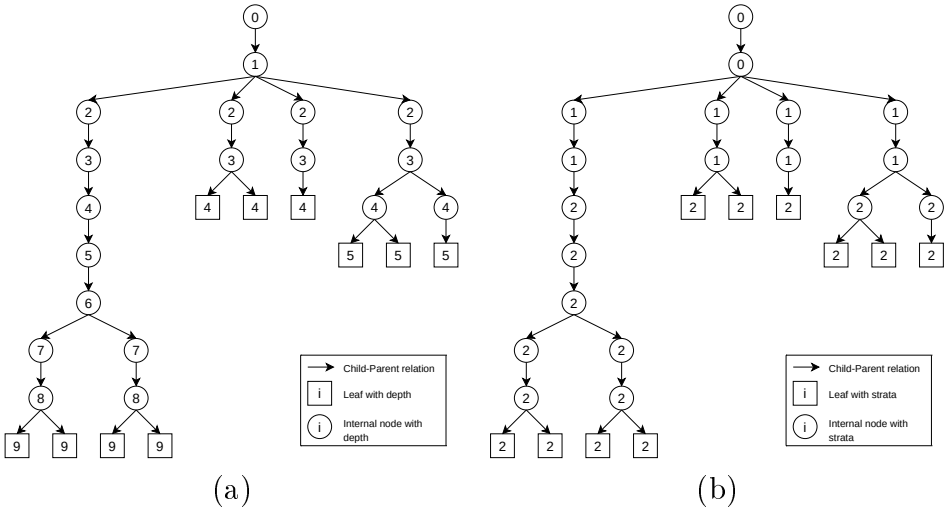


Figure 17: (a) The example trie with depths for each node and (b) the strata of each node using the root as r.

where i is the strata of node v. Figure 18 (a) shows the components of the example trie.

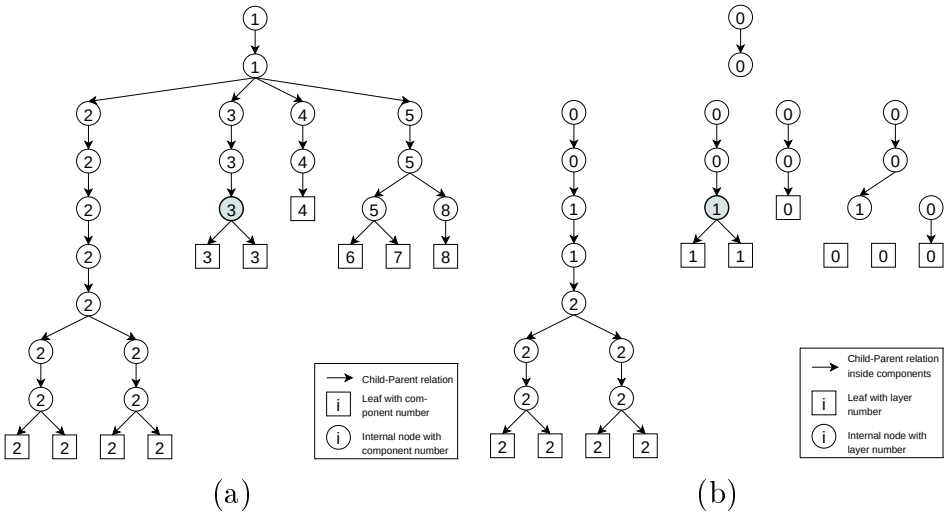


Figure 18: (a) Components of the example trie with $\varepsilon = 1$ and (b) the layers of each component. The shaded node is an inserted ε-node.

8.3 Divide a component into layers

It is straightforward to divide a component into layers. When a node is contained in a component, the strata in which it is contained becomes the layer

Strata i becomes layer i.

Figure 18 (b) shows the layer in which each node is contained. If a node has more than one child in the next layer, a dummy node[16] is inserted between the node and it children. This node will be contained in the same layer as the children. The shaded node in Figure 18 (b) is an example of a dummy node.

Let the component rooted at node v be denoted by C_v and the layer i in the subtree T_v be denoted by L_v^i. Then C_v and L_v^i can be defined as

$$
\begin{aligned}
L_v^o &= \{w \in T_v \mid \mathrm{rank}(w) = \mathrm{rank}(v) \quad \wedge \\
&\quad \mathrm{depth}(w) - \mathrm{depth}(v) < 2^{2^0}\} && (1)\\
L_v^i &= \{w \in T_v \mid \mathrm{rank}(v) - \mathrm{rank}(w) < \varepsilon 2^i \quad \wedge \\
&\quad 2^{2^{i-1}} \leq \mathrm{depth}(w) - \mathrm{depth}(v) < 2^{2^i} \quad \wedge \\
&\quad \Big(\exists u \in L_v^{i-1} : \mathrm{depth}(u) - \mathrm{depth}(v) = 2^{2^{i-1}} - 1 \wedge w \in T_u\Big)\} && (2)\\
C_v &= \bigcup_{i=0}^{\infty} L_v^i && (3)
\end{aligned}
$$

8.4 Blind tries and giraffe trees

Each layer contains one or more subtries, which all are compressed into blind tries. As these subtries are part of the original trie, nodes inside the subtrie might have children in other layers or components. When constructing a blind trie, only nodes inside the current layer of the subtrie are used. Blind tries do not have nodes in other layers or components.

In Figure 19 (a) the blind tries for component 2 of Figure 18 are shown. The blind tries will be used as a look-ahead structure, validating only parts of the search string. Validating the missing part is done in a giraffe tree, so each subtrie is covered by a forest of giraffe trees. Figure 19 (b) shows the giraffe trees for component 2. Giraffe trees have no nodes in other layers but are allowed to have references to roots in other components.

[16] An ε-node.

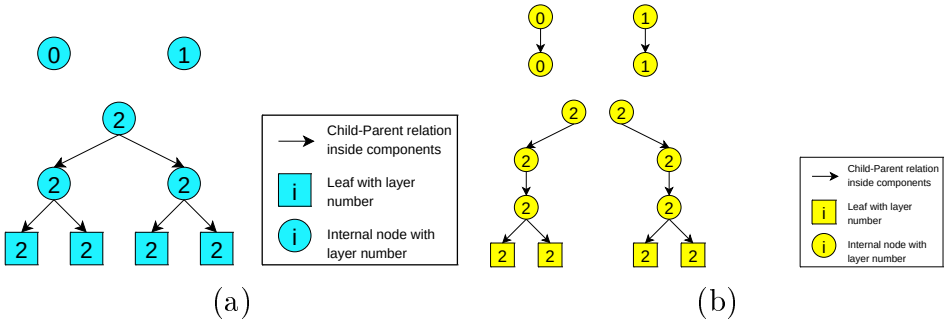


Figure 19: (a) The blind tries representing the left most component in Figure 18, i.e. component 2. (b) The giraffe trees for the same component.

In order to do a complete check of a search string, each blind trie node must have a reference to a giraffe tree covering the same trie node as the blind trie node in the original trie. In this way each time a blind trie leaf is reached, the string can be validated in the corresponding giraffe tree.

If a mismatch occurs when searching in the blind trie, it might be that the string is not inserted in the original trie. But it could also be that the search should continue in another component. As a search in a blind trie cannot be directed into another component, the giraffe trees are used. The giraffe nodes use their references to other components to direct the searches.

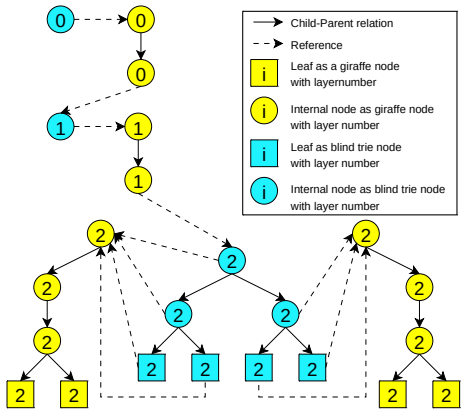


Figure 20: The blind tries and giraffe trees of component 2 put together.

Only a leaf in a giraffe tree has a reference to a blind trie. The blind trie is contained in the next layer in the same component. If a leaf is reached when validating a search string, and the string is still valid, the search is to continue in the next layer. Therefore, searching inside a component is a matter of searching in blind tries and validating in giraffe trees. Figure 20

shows the blind tries and giraffe trees of component 2 put together, enabling a search through the component.

8.5 Bridges

Giraffe nodes do not have a directly reference to roots in other components. Instead they have a reference to a weight balanced search tree[17], in which it is possible to search for the roots. When constructing the bridge, the n_v values stored in the roots are used as weights.

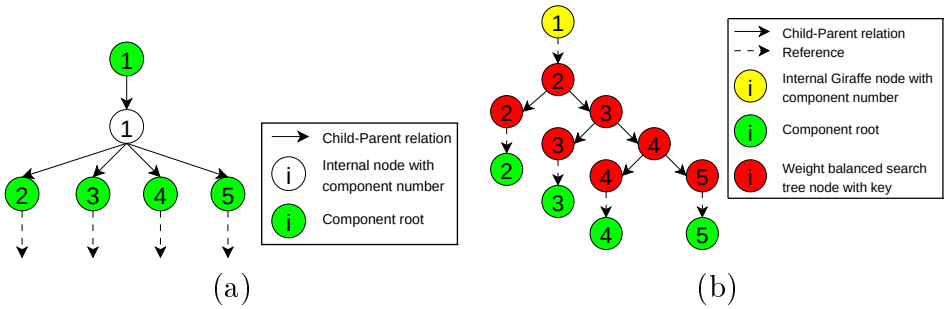


Figure 21: (a) The top of the trie where the roots of component 1, 2, 3, 4 and 5 are shown. (b) The weight balanced search tree for component roots 2, 3, 4 and 5 attached to a giraffe node from component 1.

Figure 21 (a) shows the top of the original trie, and (b) the corresponding weight balanced search tree attached to the last giraffe node of component 1. The references to component roots are in fact references to the roots of the blind tries of these components. Putting it all together, blind tries references giraffe trees. The giraffe trees references other blind tries and weight balanced search trees. The weight balanced trees references to blind tries and so on. Figure 22 shows the final structure of the example trie, when all is put together.

8.6 Component tree

The last structure presented in this section is a *component tree*, denoted T', used to do a van Emde Boas layout of the final structure, the cache oblivious string dictionary. As the van Emde Boas layout uses a binary search tree, a binary tree representing the cache oblivious string dictionary structure must be created.

[17] Also called a *bridge*.

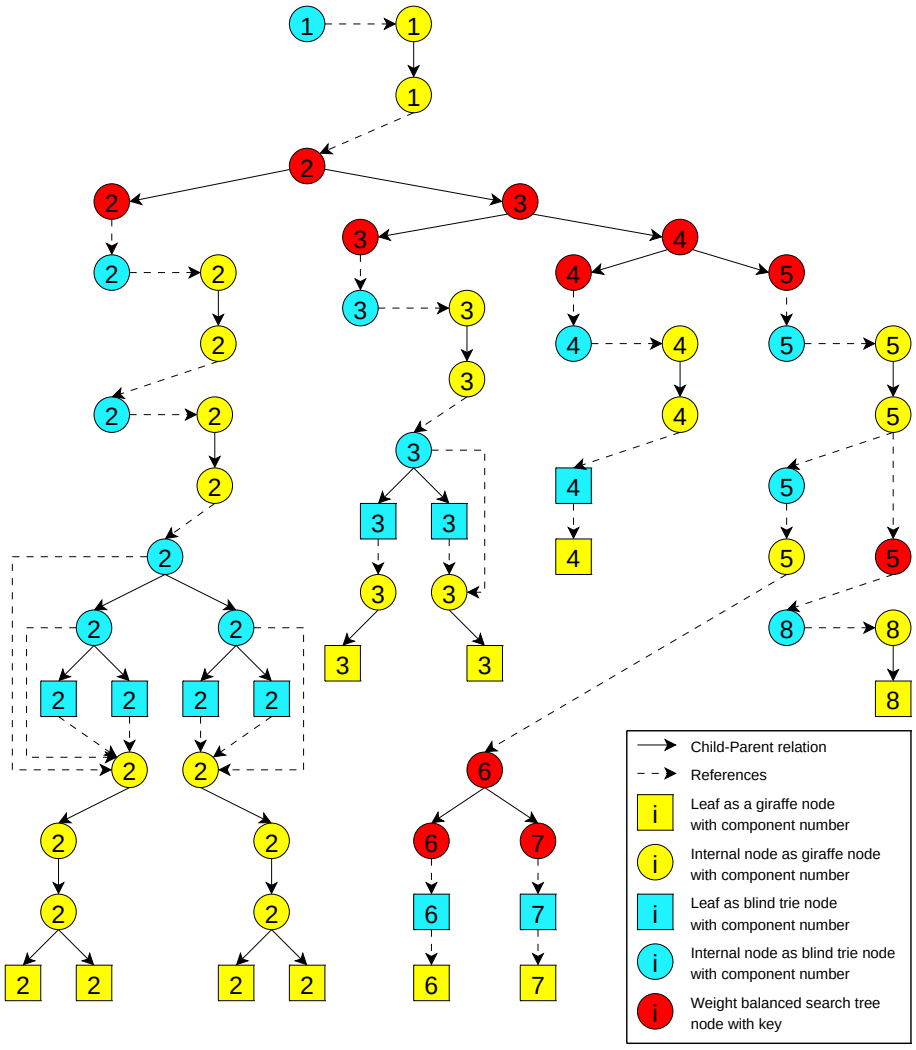


Figure 22: The final structure, where all blind tries, giraffe trees and weight balanced search trees are put together.

When creating the component tree, each component needs to identify those nodes inside the component having children in other components. These nodes are used to create a binary weight balanced tree[18] representing the component. The weight of a node, when creating the weight balanced tree, is the sum of the n_v values of its children located in other components. A binary weight balanced tree could be created by the Huffman algorithm, the article algorithm.or even the leaf oriented optimal binary search tree algorithm.

The weight balanced trees representing the components are glued together using the bridges[19] constructed earlier. This is done by connecting the nodes having children in other components with the corresponding roots of the bridges. Figure 23 (a) shows the component tree for the final structure in Figure 22. In this example all the trees representing components only contains a single node[20]. The red nodes are the same as in both figures.

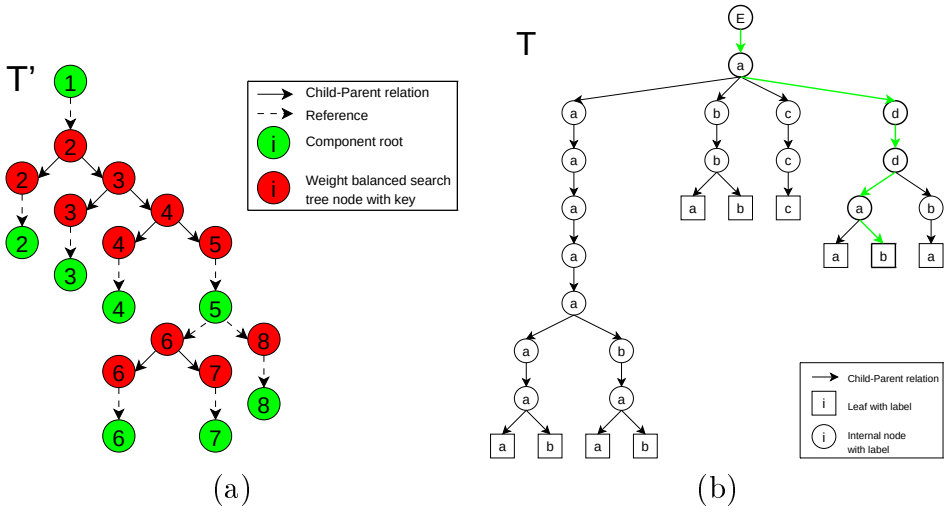


Figure 23: (a)The complete component tree for the example structure. (b) The example trie with labels inserted. The green path highlight a search for the string addab. Nodes who label is E is epsilon (dummy) nodes.

8.7 Search example

To illustrate how different the trie structure is from the cache oblivious string dictionary structure in Figure 22, a short search example is presented. Fig-

[18]Note it need not be a search tree.
[19]The weight balanced search trees.
[20]These are the green nodes.

ure 23 (b) shows the original trie with labels inserted into the nodes. The green path is the search path when searching for the string `addab`. The ε-node at the top is just a dummy node used as a starting point.

The same search is shown in Figure 24 where the blind tries, giraffe trees and weight balanced search trees are traversed. The nodes uses the same labels as in the original trie. As seen, there is a great difference in the search patterns.

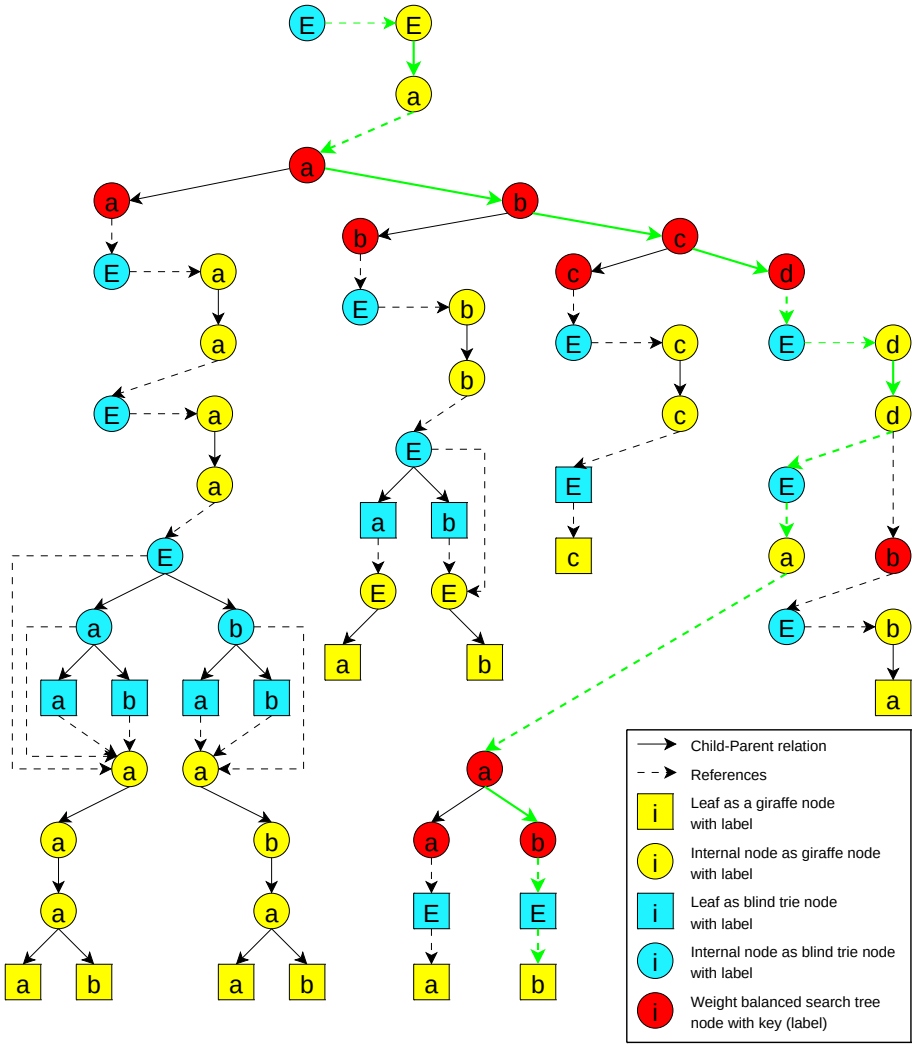


Figure 24: A search for the string addab in the cache oblivious string dictionary structure. The path is highlighted with green, showing which nodes are traversed during the search. A node labelled with an E indicates an ε-node.

9 Memory layout

A van Emde Boas layout of the component tree T' is not trivial, as the structure contains blind tries, giraffe trees and weight balanced search trees. The van Emde Boas layout of the component tree is used to determine in what order the different component are laid out in memory, while the depth of the recursion determines the order of the layers.

9.1 Layout of component tree T'

There are two kind of trees in the component tree. The first is the weight balanced tree induced by the bridge nodes in each component. The second is the weight balanced search tree connecting the components. The layout of a node depends on which kind of tree it belongs to.

In a weight balanced tree, only the root is laid out in memory. The rest are considered dummy nodes. Doing a layout of the root, means doing a layout of the blind trie and the associated giraffe trees, located in the first layer[21] of the component. The rest of the component, layer $1, 2, \ldots, k$, are laid out later, Section 9.2. The dummy nodes are ignored and thus not laid out.

The nodes in a weight balanced search tree do not represent any components and are simply laid out when reached in the recursion.

Figure 26 shows the recursive call on the top of the component tree from Section 8, Figure 25. In the figure the layout at the bottom is only a pseudo layout illustrating the order of the nodes. Only the component roots are shown, hiding the details of the blind tries and giraffe nodes. The triangles represent the different recursions of the layout.

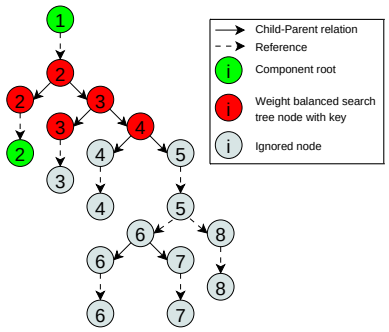


Figure 25: The top of the component tree from Section 8.

[21] Layer 0.

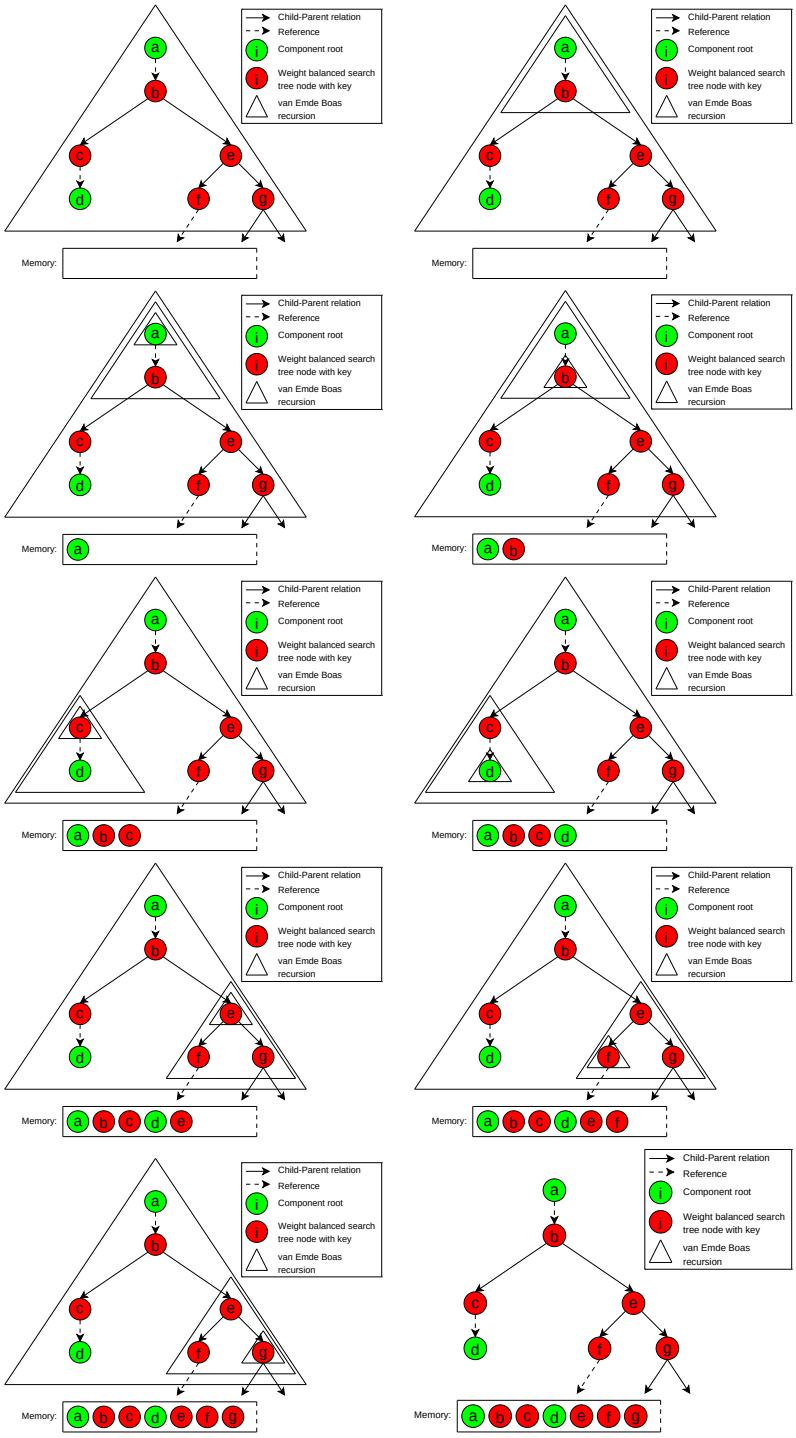


Figure 26: The recursions of a van Emde Boas layout on the top of the example trie from Section 8. Instead of component numbers, each node has been given a letter, making it easier to identify it in the memory layout

9.2 Depth of recursion

In a van Emde Boas layout of a component tree, the depth of the recursion is used to determine in which order to lay out the different layers of the components. The depth of the recursion is numbered in reverse order, beginning with the inner most recursive call. This is illustrated in Figure 27.

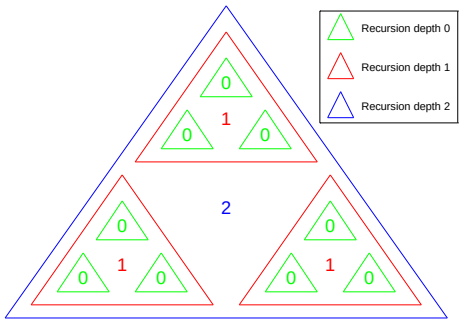


Figure 27: The depth of recursive calls is numbered in reverse order, starting with the inner most recursive call.

The reason for this is that the depth now correspond to the layers of the components when these are laid out. Roughly speaking, when a component[22] is contained in a recursion of depth 0, then its layer 0 is laid out. When it is contained in depth 1, its layer 1 are laid out and so on. The layers are always laid out in increasing order, so layer 0 is the first to be laid out, then layer 1 and so forth. Put in another way, when a components ith layer is laid out, its $(i+1)$th layer is laid out in the recursive call it returns to.

To illustrate this, Figure 28 shows an example of this. Starting with a recursive call at depth 2, the tree is divided into one top and two bottom trees. The top is recursively divided again into a top node and a bottom node (a). As component 1 is contained in a recursive call at depth 0, height of the tree is 1, its layer 0 is laid out. Following the van Emde Boas layout, the bottom node is visited and laid out, as it is a weight balanced search tree node, (b). Now the call returns to recursion depth 1 of the top tree, and all components contained inside the top triangle will have their next layer laid out. In this case layer 1 of component 1, (c).

Next in the layout, the left bottom tree is visited, and its top node laid out, (d). Then layer 0 of component 2 is laid out, as it is visited at recursion depth 0, (e). Returning to recursion depth 1 of the left bottom tree, layer 1 of component 2 is laid out, (f). The same happens to the right bottom tree, (g), (h), (i). Returning from depth 1 of the bottom right tree, the recursion

[22]By component means the root of the weight balanced tree.

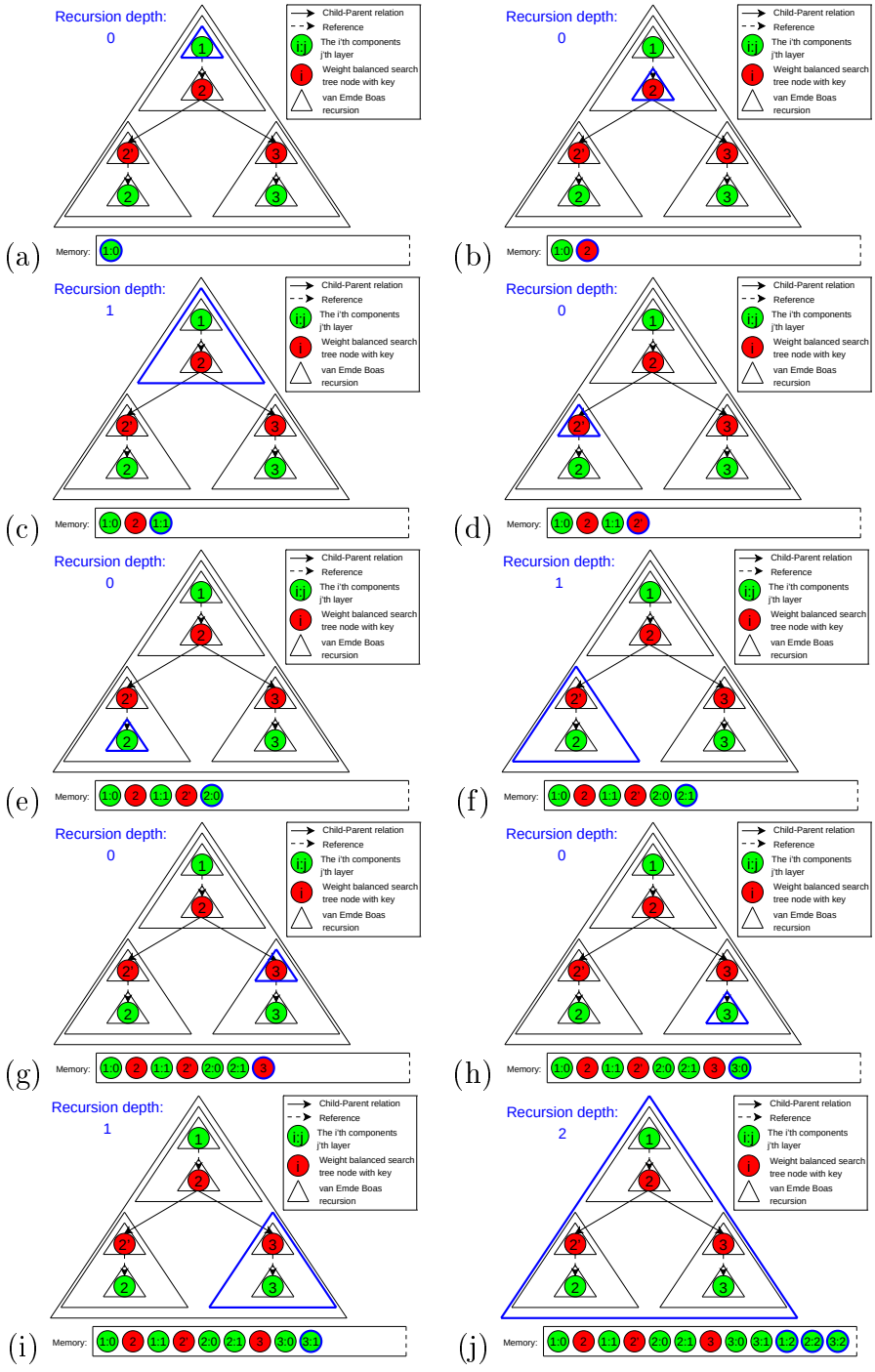


Figure 28: Pseudo layout of a tree showing the order of the different layers.

returns to depth 2, which covers all the components. Therefore, their next layer is laid out, (j). As no more recursive call have been made, i.e. there is no recursion of depth 3, the rest of the components layers (if any) are laid out.

9.3 Blind trie and giraffe tree layout

When doing a layout of a layer, all the blind tries contained in the layer followed by the associated giraffe trees are laid out. Both a blind trie and a giraffe tree is laid out top to bottom in BFS [23] order. Figure 29 shows an example, taken from section 8. The labels are for illustrating purpose only.

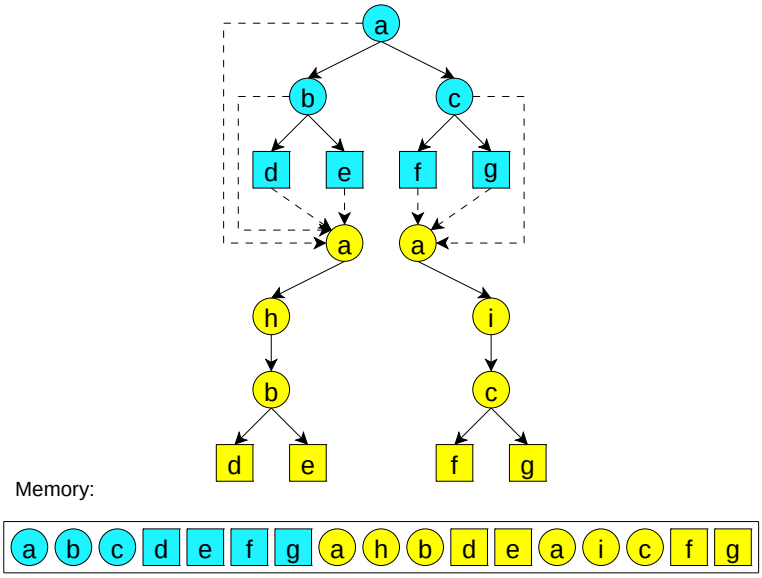
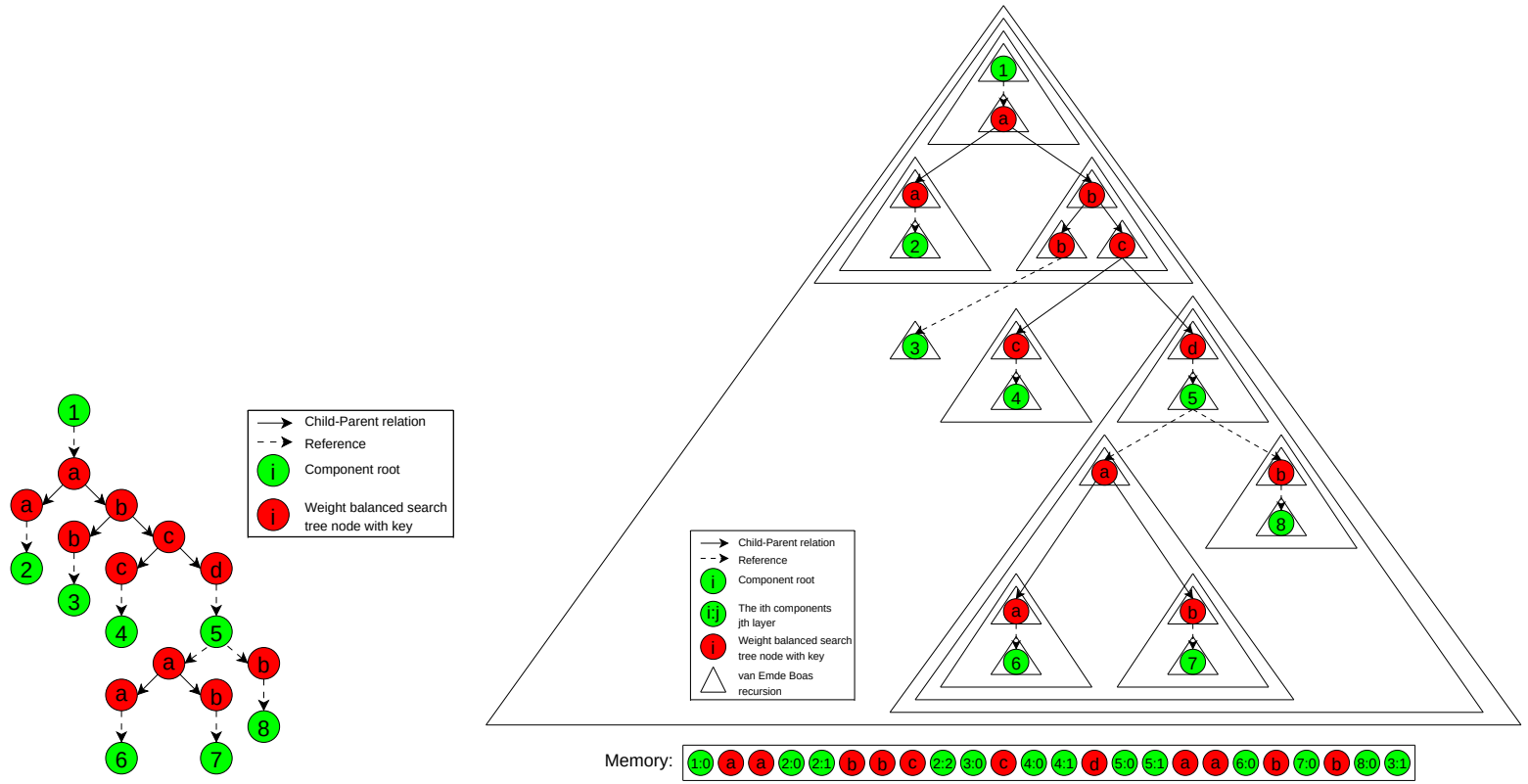


Figure 29: The layout of layer 2 of component 2. First the blind trie is laid out in BFS order followed by the two giraffe trees associated with the blind trie, also in BFS order.

9.4 Layout example

To complete the example from Section 8 a complete layout of the cache oblivious string dictionary structure is given. Figure 30 (a) shows the component tree, (b) the van Emde Boas recursions and Figure 31 the structure and the layout.

[23] Breadth First Search



(a) (b)

Figure 30: (a) The component tree from Section 8 and (b) the van Emde Boas recursions of the component tree with the pseudo layout. Non-existing layers are removed from the layout.

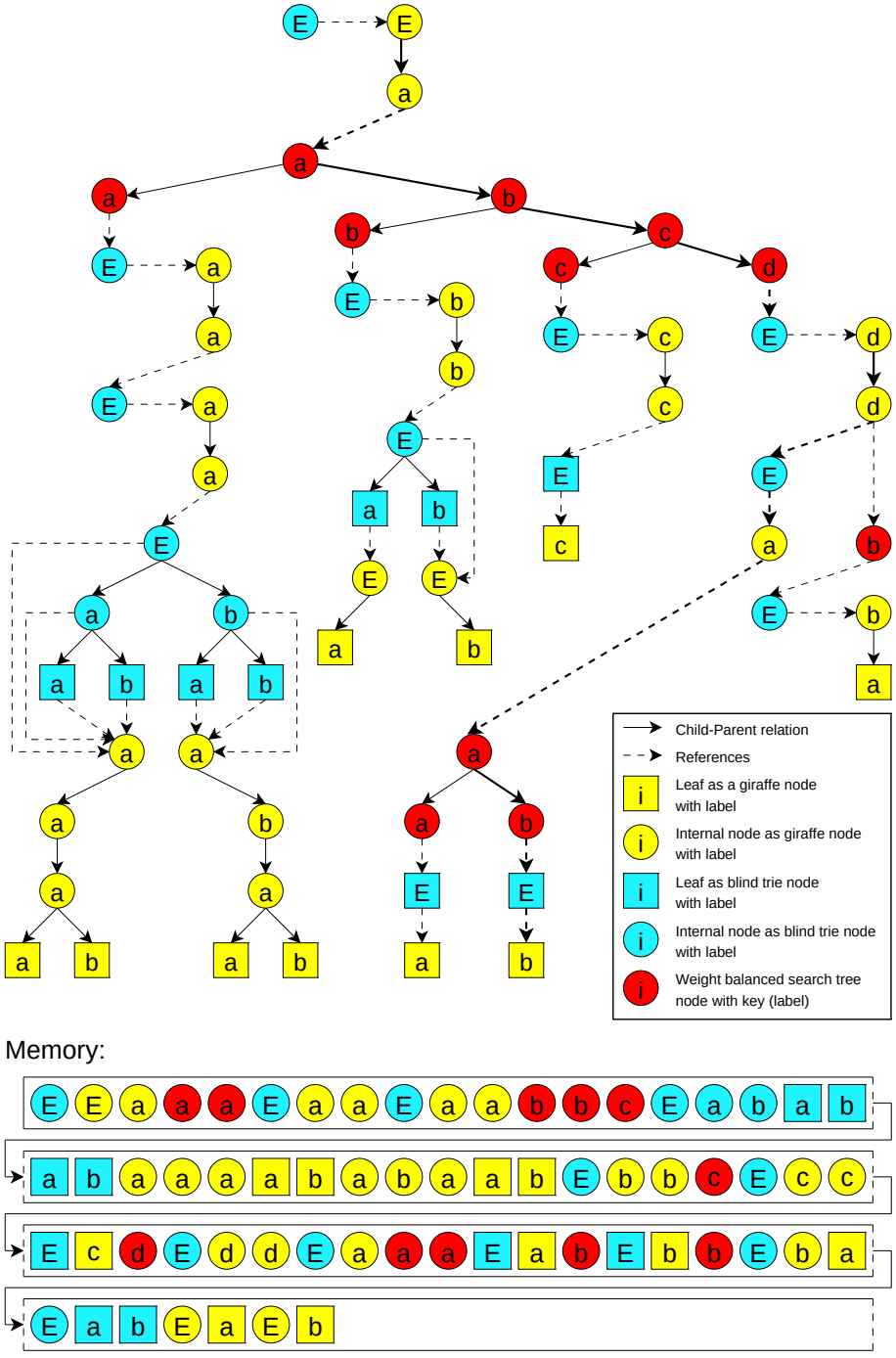


Figure 31: The whole structure of the example tree and its layout using van Emde Boas layout in Figure 30. The label E indicates an ε-node.

10 Searching

The search for a string s is done in three different structures, the blind trie structure, the giraffe tree structure and the weight balanced search tree structure. Validating whether or not a string is present in the cache oblivious string dictionary requires traversal of these structures.

When searching for a string s in a cache oblivious string dictionary structure, the first part of the string is checked in the blind trie located at the root. This results in a verification in a giraffe tree, which directs the search for the next part of s into another blind trie, possibly using a weight balanced search tree. This continues until a mismatch occur or s is found to be contained.

10.1 Searching in a blind trie

A search for a string s in a blind trie is done top-down. When reaching a node the label at that node is not compared to the corresponding string.

Instead the children are searched. As the reached node contains how many characters there have been omitted, it is possible to find the character in s to which at most one of the children must match. As the children have different labels, only one of the children can match the corresponding character in s. Therefore, the search in the blind trie is unique.

When a leaf is reached, the giraffe tree associated with the leaf is returned. This giraffe tree contains all characters between the blind trie root and the leaf, including all the omitted characters. Therefore, it can be used to do a complete check of the part in s checked by the blind trie.

If a mismatch is found at a node, i.e. none of the children matches, its corresponding giraffe tree is returned. The reason is that the search to the parent corresponded to s, but the search from parent to child did not. It is possible that somewhere between the parent and child, the search needed to continue in another component. This can only be verified by traversing the giraffe tree, as the blind trie is constructed of the internal nodes in the layer.

10.2 Searching in a giraffe tree

Searching in a giraffe tree is similar to searching in a blind trie. The search is done top-down, and since no characters were omitted when the giraffe tree was constructed s can be fully checked.

The result of a search in a giraffe tree is either a blind trie root or nothing. A search is only continued into a child node if the parent matches the corresponding character in s, and the child matches the next character in s.

If the parent matches and none of the children does, the search cannot continue in the giraffe tree. Then the weight balanced search tree located at the node is searched. If the search is successful, the blind trie reached in the weight balanced search tree is returned. If not the search is ended as s did not have a match in the giraffe tree.

When a giraffe leaf is reached, the weight balanced search tree is searched for the next character in s. If a match is found the blind trie from this search is returned. Otherwise, the blind trie referenced to from the giraffe leaf (if any) is returned.

10.3 Searching in a weight balanced search tree

A weight balanced search tree is traversed as any other search tree. At each node the label stored at the node is checked against the corresponding character in s, and the search continues into the left or right child. When a leaf is reached, the leafs label is compared with the corresponding character in s. If it is a match, the blind trie referenced at the leaf is returned.

10.4 Example of searching

Returning to the search example in Figure 24 from Section 8 it is now possible to describe the search in details.

1. First the blind trie at the root is searched for the character `a`. The ε-node matches everything so the search continues. As the node is a leaf, the giraffe tree referenced at the node is returned

2. The giraffe tree is used for checking the match of `a`. As an ε-node matches everything, the search proceeds into the child, where a match is found. As the child is a leaf, the search is directed to the weight balanced search tree, searching for the letter `d`. A match is found to the right[24] and the blind trie whose root contains the character `d` is returned.

3. The blind trie is searched for the character `d`. As the blind trie consists of just an ε-leaf the giraffe tree associated are returned. The giraffe tree is used for validating the result. The blind trie referenced from the giraffe tree leaf is returned.

[24] In the example drawing.

4. The search in the blind trie and giraffe tree matches the character `a` and the search continues into the weight balanced search tree at the bottom, where a match is found.

5. The blind trie returned from the weight balanced search tree is searched for the letter `b`. Again the blind trie is an ε-node and the checking in the giraffe tree is successful. As s contains no more characters, the giraffe node is checked for any strings ending at this node in the original trie. This information is stored in the giraffe node, and this result is returned ending the search.

11 Analysis

The analysis is divided into two part. The first part analyses the space usage of the cache oblivious string dictionary structure and the second part analyses the search time. Before analysing the space usage a small lemma and theorem is presented, setting an upper bound on the number of leaves in a layer, and the number of components in the structure.

Lemma 11.1

1. *If a node $w \in C_v$ has a child u with rank(u) = rank(w), then w and u are in the same component.*
2. *If a node $w \in T$ has only one child u, then w and u are in the same component.*
3. *L_v^i is a forest with at most $2^{\varepsilon 2^i+1}$ leaves.*
4. *L_v^i contains at most $(2^{2^i} - 2^{2^{i-1}})2^{\varepsilon 2^i+1}$ nodes.*
5. *For a node $w \in L_v^i, w \neq v$, with a child $u \notin C_v$, then rank$(v)-$rank$(u) \geq \varepsilon 2^i$.*

Proof

1. Since $\text{rank}(v) - \text{rank}(w) < \varepsilon 2^i$ so is $\text{rank}(v) - \text{rank}(u)$ and thus the child u is a candidate, if w is a candidate. (The candidate requirement is $\text{rank}(v) - \text{rank}(w) < \varepsilon 2^i$ for a node $w \in L_v^i$)
2. Same argument as (1). Since they both have the same number of leaves, $n_w = n_u$, they have the same rank.
3. First let $w_1, w_2, \ldots, w_k$ be leaves of L_v^i. A leaf is a node in L_v^i having no child in L_v^i. As the subtrees $T_{w_1}, T_{w_2}, \ldots, T_{w_k}$ (of T_v) are disjoint then $n_{w_1}, n_{w_2}, \ldots, n_{w_k} \leq n_v$. From the candidate requirement, it follows that for a leaf $w_j, 1 \leq j \leq k$, it is know that

$$\begin{aligned}
\text{rank}(v) - \text{rank}(w_j) &\leq \varepsilon 2^i \Rightarrow \\
\lceil \log(n_v) \rceil - \lceil \log(n_{w_j}) \rceil &\leq \varepsilon 2^i \Rightarrow \\
2^{\lceil \log(n_v) \rceil} - 2^{\lceil \log(n_{w_j}) \rceil} &\leq 2^{\varepsilon 2^i} \Rightarrow \\
\frac{2^{\lceil \log(n_v) \rceil}}{2^{\varepsilon 2^i}} &\leq 2^{\lceil \log(n_{w_j}) \rceil} \Rightarrow
\end{aligned}$$

$$\begin{aligned} \frac{n_v}{2 \cdot 2^{\varepsilon 2^i}} = \frac{n_v}{2^{\varepsilon 2^i + 1}} &\leq n_{w_j} \Rightarrow \\ \frac{n_v}{n_{w_j}} &\leq 2^{\varepsilon 2^i + 1} \end{aligned}$$

from which it can be concluded, that L_v^i has at most $2^{\varepsilon 2^i + 1}$ leaves.

4. Let w be a leaf in L_v^i. It has at most $2^{2^i} - 2^{2^{i-1}}$ ancestors in L_v^i since it fulfil the strata requirement $2^{2^{i-1}} \leq \text{depth}(w) - \text{depth}(v) < 2^{2^i}$. As there can be at most $2^{\varepsilon 2^i + 1}$ leafs in L_v^i, (3), there are at most $(2^{2^i} - 2^{2^{i-1}})2^{\varepsilon 2^i + 1}$ nodes in L_v^i.

5. Let $w \in L_v^i$, $w \neq v$ and let $u \notin C_v$ be a child of w. Since $u \notin C_v$ this mean, that u does not fulfil the candidate requirement, and hence $\text{rank}(v) - \text{rank}(u) \not< \varepsilon 2^i \Rightarrow \varepsilon 2^i \leq \text{rank}(v) - \text{rank}(u)$

□

Theorem 11.1 *On a root-to-leaf path in a tree T with n leaves, there are at most $1 + \lceil \log(n) \rceil$ components*

Proof As the ranks of the component roots are strictly decreasing, lemma 11.1 (1), there can be at most $\lceil \log(n) \rceil + 1$ components, as the first has rank $= \log(n)$, the second rank $= \log(n-1)$ and so on.

□

11.1 Space usage

The space usage of a blind trie covering the trie T is dominated by the space usage of the associated covering of T by giraffe trees. Therefore, it is sufficient to look at the space usage of this covering of giraffe trees.

Lemma 11.2 *The algorithm in Figure 6 constructs a covering of T with giraffe trees of total size $O(N)$ where N is the number of nodes in T*

Proof Let $T^{i:j}$ and $T^{j+1:k}$ be two consecutive giraffe trees constructed using the algorithm in Figure 6. Observe that the only nodes from $T^{i:j}$ which can appear in any succeeding giraffe tree constructed after $T^{i:j}$ are those on the path to leaf l_{j+1}, i.e. the rightmost ones.

For the construction of $T^{i:j}$ two sets, $A^{i:j}$ and $B^{i:j}$ will be charged. $A^{i:j}$ is the set of nodes in $T^{i:j}$ which is not on the path to l_{j+1}. $B^{i:j}$ is the set

of nodes on the path to l_{j+1} but not on the path to l_i. Figure 32 shows two cases of these sets[25].

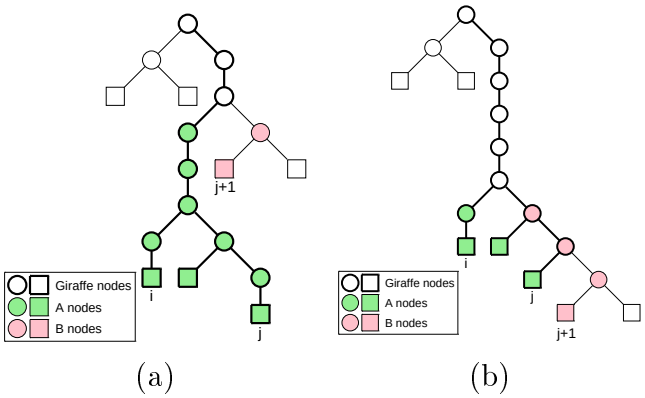


Figure 32: Two cases showing the sets $A^{i:j}$ and $B^{i:j}$.

As $T^{i:j}$ and $T^{j+1:k}$ are two consecutive giraffe trees, this means that $T^{i:j+1}$ cannot be a giraffe tree, i.e. its neck is too short. Removing all the nodes in $A^{i:j}$ and $B^{i:j}$ from $T^{i:j+1}$ leaves only the section of the neck shared by $T^{i:j}$ and $T^{i:j+1}$. This implies[26]

$$|A^{i:j}| + |B^{i:j}| > \frac{|T^{i:j+1}|}{2}$$

and

$$|T^{i:j}| < |T^{i:j+1}| < 2\left(|A^{i:j}| + |B^{i:j}|\right)$$

A node is only contained in $A^{i:j}$ exactly once, which is in the last giraffe tree using it. Similar a node is contained at most once in $B^{i:j}$. This is when it is used in the tree constructed prior to the tree, where it is contained in the leftmost path for the first time. Notice, for the last tree to be constructed, $T^{k:n}$, no leaf l_{n+1} exists, therefore, $A^{k:n} = T^{k:n}$. From this follows

$$\sum_{T^{i:j}} |T^{i:j}| < \sum_{T^{i:j}} 2\left(|A^{i:j}| + |B^{i:j}|\right) \leq 4N$$

□

In order to analyse the space usage of a subtree of the van Emde Boas layout, the height of the weight balanced tree is needed, as this is used in the

[25]Both taken from [Brodal and Fagerberg, 2006].
[26]Because of the short neck.

component tree. The algorithm used to construct the weight balanced tree is analysed in the following lemma.

Lemma 11.3 *Let $x_1 \leq x_2 \leq \ldots \leq x_n$ be a list of n keys in sorted order, and let each key x_i have an associated weight $w_i \in \mathbb{R}_+$. Let $W = \sum_{i=1}^{n} w_i$. The algorithm in figure 11 constructs a binary search tree where each key x_i is contained in a leaf of depth at most $2 + 2\lceil \log(W/w_i) \rceil$.*

Proof Let the rank of a node be the rank of the tree rooted at the node. Denote an edge *efficient* if the rank of its upper node is larger than the rank of its lower node. Let an inefficient edge be *covered* if the edge immediately after is efficient. To see there are k efficient edges in a root to leaf path each linking in the algorithm must be examined.

Consider the linking of trees on the stack S including st (Lowest tree in S for which $\text{rank}(st) \leq \text{rank}(T')$). As all trees in S have different rank, and the ranks are decreasing from bottom to top, each linking of the two top trees will contain at least one efficient edge, Figure 33. This way of linking insures that all inefficient edges are covered (Except possibly edges to the root and incident to leaves).

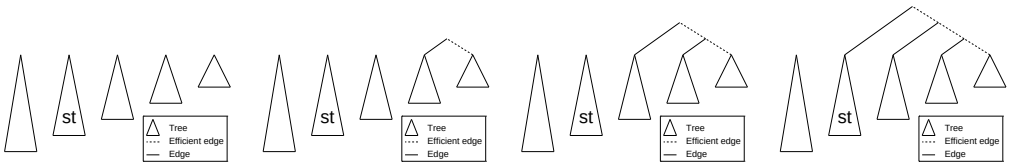


Figure 33: Linking trees from the stack results in all inefficient edges will be covered.

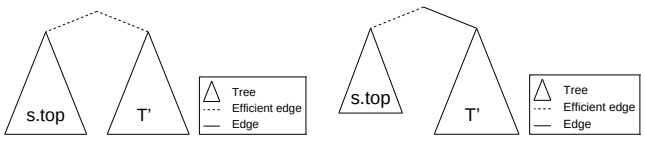
After the linking up until st, three cases exists:

rank$(T') <$ **rank**$($**S.top**$)$ As T' is just pushed onto the stack S, no linking involving T' happens.

rank$(T') =$ **rank**$($**S.top**$)$ The first linking of T' and S.top will result in two efficient edges, as the new tree will have $\text{rank}(T') + 1$, Figure 34 (a).

rank$(T') >$ **rank**$($**S.top**$)$ The linking of S.top and T' results in one efficient edge, Figure 34 (b).

Since there are k efficient edges in a root-to-leaf path, there can be at most $2k + 2$ edges including edges to the root and incident to leaves. Going from the root to a leaf containing key x_i, it follows that



(a) (b)

Figure 34: Linking S.top with T' results in at least one efficient edge.

$$\begin{aligned}
\lceil \log(w_i) \rceil + k &\leq \lceil \log(W) \rceil \Rightarrow \\
log(w_i) + k &< 1 + \log(W) \Rightarrow \\
k &< 1 + \log\left(\frac{W}{w_i}\right) \Rightarrow \\
k &\leq \left\lceil \log\left(\frac{W}{w_i}\right) \right\rceil
\end{aligned}$$

as k is an integer.

□

Now is it possibly to analyse the height of the component tree T'. This is done in theorem 11.2.

Theorem 11.2 *The height of T' is $O(\log(n))$ where n is the number of leaves in T.*

Proof A root to leaf path in T corresponds to a root to leaf path in T'. The number of components traversed in a root to leaf path is $O(\log(n))$, theorem 11.1. Each of these components are replaced by a weight balanced tree in T'. In a root to leaf path in T' the number of nodes visited in the weight balanced tree for component i is $2 + 2\log(w_{i+1}/w_i)$, lemma 11.3. Therefore, the total number of nodes visited is

$$\begin{aligned}
O\left(\sum_{i=0}^{\log(n)} 2 + 2\left\lceil \log\left(\frac{w_{i+1}}{w_i}\right)\right\rceil\right) &= O\left(2\log(n) + 2\sum_{i=0}^{\log(n)}\left(1 + \log\left(\frac{w_{i+1}}{w_i}\right)\right)\right) \\
&= O\left(4\log(n) + \sum_{i=0}^{\log(n)} \log\left(\frac{w_{i+1}}{w_i}\right)\right) \\
&= O\left(4\log(n) + \log\left(\frac{w_{\log(n)}}{w_0}\right)\right) \\
&= O\left(\log(n)\right)
\end{aligned}$$

using the telescope property ($\log(x/y) + \log(y/z) = \log(x/z)$).

□

As the height of the component tree is bounded, only the space required for each layer is missing. The space usage for layer i is found in lemma 11.4.

Lemma 11.4 *Storing L_v^i uses $O(|L_v^i|)$ space, which is $O(2^{2^{i+1}})$.*

Proof From lemma 11.2 the total space required for L_v^i is $O(N)$. By Lemma 11.1 (4) this is dominated by $O((2^{2^i} - 2^{2^{i-1}})2^{\varepsilon 2^i})$. Since $\varepsilon \leq 1$ this is $O(2^{2^{i+1}})$

□

It is now finally possibly to analyse the space usage of a subtree X of T'.

Theorem 11.3 *A subtree X of T' of height 2^i in the van Emde Boas layout of T' requires space $((2^{2^i})^3)$.*

Proof As T' is a binary tree, so is X. Since the height of X is 2^i, it contains at most 2^{2^i} leaves, Figure 35.

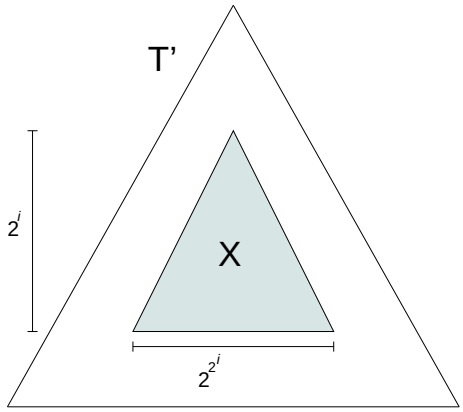


Figure 35: The subtree X inside the binary tree T'.

Therefore, X contains $O(2^{2^i})$ component nodes, each having their layer $0, 1, \ldots, i = \log(2^i)$ inside X. The rest of the layers are placed outside X. As each layer uses $O(2^{2^{i+1}})$ space, lemma 11.4, the space usage is $O(2^{2^i} \cdot \sum_{j=0}^{i} 2^{2^{j+1}})$. In the sum, layer i dominates the previous layers, implying the space usage is

$$
\begin{aligned}
O\left(2^{2^i} \cdot \sum_{j=0}^{i} 2^{2^{j+1}}\right) &= O\left(2^{2^i} \cdot 2^{2^{i+1}}\right) \\
&= O\left(2^{2^i} \cdot 2^{(2^i)2}\right) \\
&= O\left(2^{2^i} \cdot \left(2^{2^i}\right)^2\right) \\
&= O\left(\left(2^{2^i}\right)^3\right)
\end{aligned}
$$

□

11.2 Time usage

Traversing the cache oblivious string dictionary is a matter of traversing blind tries and giraffe trees. The following theorem bounds the I/Os of traversing a blind trie, compared to the previous traversed giraffe tree.

Theorem 11.4 *The number of I/Os that may be done in traversing the pattern while searching in the blind trie for the given ith layer is at most a constant factor greater than the number of I/Os done in traversing the giraffe trees for the previous layer*

Proof An ith layer contains at most $2^{\varepsilon 2^i+1}$ leaves, lemma 11.1 (3), searching the associated blind trie takes at most $O(2^{\varepsilon 2^i+1})$ I/Os. The root-to-leaf path of the giraffe tree from the previous layer contains at most $O(2^{2^{i-1}})$ nodes, thus traversing the giraffe from the previous layer takes at most $O(2^{2^{i-1}}/B)$ I/Os.

□

Since traversing a blind trie is only a constant factor greater than traversing the previous giraffe tree, it is interesting to bound the number of I/Os used traversing a giraffe tree.

Lemma 11.5 *Let T be a giraffe tree with N nodes stored in BFS layout. Traversing a path of length p starting at the root of T requires $O(p/B)$ I/Os.*

Proof There exists two cases. If $p < N/2$ then p is contained in the topmost nodes of the giraffe, which all are laid out consecutively left to right. Therefore, accessing the path requires $O(p/B)$ I/Os. Otherwise the path might go

from the root to a leaf. As the nodes are laid out in BFS that is they are laid left to right in memory, following the path is bounded by scanning the array containing all the nodes $O(N/B) = O(p/B)$ I/Os.

□

The previous lemma leads to the following theorem.

Theorem 11.5 *Given a tree T with N nodes, there exists a cache oblivious covering of T by subtrees (giraffe trees) where the total space requirement of the covering is $O(N)$, each root-to-leaf path is present in one subtree and the prefix of length p of a predetermined root-to-leaf path can be traversed in $O(p/B)$ I/Os.*

Proof This follows directly from lemma 11.5 and 11.2. As each leaf is charged as an $A^{i:j}$ exactly once, each root-to-leaf path is present in one subtree.

□

Finally it is possible to bound the number of I/Os used, when searching in a cache oblivious string dictionary.

Theorem 11.6 *Prefix queries for a query string P in a string dictionary storing n strings use $O(\log_B(n) + |P|/B)$ I/Os.*

Proof The number of I/Os used in a search in the cache oblivious string dictionary are caused by either accessing the search string P or by accessing the string dictionary structure. First the number of I/Os used when accessing P is analysed and second accessing the cache oblivious string dictionary structure.

Scanning P from left to right takes $\lceil |P|/B \rceil$ I/Os. Unfortunately, the blind trie uses random I/Os when looking ahead in P, so extra care need to be taken to bound the number of random I/Os. Assume without loss of generality, that the next $\Theta(M)$ unmatched characters of P are kept in memory. Only look-ahead of $\Omega(M)$ characters can now cause a random I/O.

Consider the case where an access to L_v^i causes a look-ahead of $\Omega(M)$ during the blind trie search for L_v^i, i.e. $\Omega(M) = 2^{2^i}$. As L_v^i has size $O(2^{\varepsilon 2^i})$, lemma 11.1 (3), and thereby $O(2^{\varepsilon 2^i})$ possibly random I/Os, then in order for the match characters, $\Omega(2^{2^{i-1}})$, in the previous layer L_v^{i-1} to pay for the random I/Os then

$$B \cdot 2^{\varepsilon 2^i} = O(2^{2^{i-1}}) \tag{4}$$

is needed.

Assuming a tall cache assumption $B^{2+\delta} \leq M$ for some constant $\delta > 0$, (4) can be shown. Using the assumption

$$B \leq M^{\frac{1}{2+\delta}} \leq \left(2^{2^i}\right)^{\frac{1}{2+\delta}}$$

since $M \leq 2^{2^i}$. Using this, it follows that

$$\begin{aligned}
\left(2^{2^i}\right)^{\frac{1}{2+\delta}} \cdot 2^{\varepsilon 2^i} &\leq 2^{2^{i-1}} \Rightarrow \\
2^{2^i \cdot \frac{1}{2+\delta} + \varepsilon 2^i} &\leq 2^{2^{i-1}} \Rightarrow \\
2^i \cdot \frac{1}{2+\delta} + \varepsilon 2^i &\leq 2^{i-1} \Rightarrow \\
2^i \left(\frac{1}{2+\delta} + \varepsilon\right) &\leq 2^i \cdot \frac{1}{2} \Rightarrow \\
\frac{1}{2+\delta} + \varepsilon &\leq \frac{1}{2} \Rightarrow \\
\varepsilon \leq \frac{1}{2} + \frac{1}{2+\delta}
\end{aligned}$$

showing, that for a constant $\delta > 0$, with the corresponding ε, then the matched characters in P in L_v^{i-1} can pay for the random I/Os caused by look-ahead in P for layer L_v^i.

For counting the I/Os caused by accessing T', a search path in T' is considered. From theorem 11.3 it follows that a subtree of height 2^t in the van Emde Boas layout, containing the t first L_v^i, uses space $O((2^{2^t})^3)$. Assuming that the currently traversed height 2^t subtree is always kept in memory, then

$$\left(2^{2^t}\right)^3 \leq B \Rightarrow 2^t \leq \frac{1}{3}\log(B)$$

If the search path only searches the t first layers in each component, then a root-to-leaf search in T' will cause

$$\frac{\log(n)}{\frac{1}{3}\log(B)} = 3\log_B(n) = O(\log_B(N))$$

I/Os.

To account for the rest of the layers $L_v^{t+1}, L_v^{i+2}, \ldots, L_v^s$, a little more is needed. Assume without loss of generality that each $L_v^i, (t < i < s)$, needs to be read into memory. Using lemma 11.1 (3) and theorem 11.5 each of these needs

$$O\left(\frac{2^{\varepsilon 2^i}}{B} + \frac{p_i}{B} + 1\right)$$

I/Os, where p_i is the length of the path matched in L_v^i. For

$$\varepsilon \leq \frac{1}{2} \Rightarrow 2^{\varepsilon 2^i} \leq 2^{2^{i-1}}$$

the scanning of the blind trie for L_v^i is dominated by the matched part of L_v^{i-1}

$$O\left(\frac{2^{\varepsilon 2^i}}{B}\right) = O\left(\frac{p_{i-1}}{B}\right)$$

making the total number of I/Os

$$O\left(\sum_{i=t+1}^{s} \frac{p_i}{B} + 1\right)$$

It now remains to charge the $+1$ in the sum, as p_i/B are the cost of scanning p_i. Since $(2^{2^{t+1}})^3 = 2^{2^{t+3}} = \Omega(B)$ and $p_i = \Theta(2^{2^i})$ this implies that the two layers, $t+1$ and $t+2$, might not fill a block B fully, i.e. $p_i/B = o(1)$. The same applies for layer s, as it might not be searched fully. For the rest of the layers $t+3, t+4, \ldots, s-1$, the $+1$ can be charged p_i/B as $p_i/B = \Omega(B)$, i.e. the search in the giraffe for L_v^{i+1} pays for the search in the blind trie in L_v^i.

From lemma 11.1 (5) it follows that the rank decreases by at least $\varepsilon 2^k$ when changing component at $L_v^k, t < k$. As $(2^{2^{t+1}})^3 = \Omega(B)$ this implies that $\varepsilon 2^k = \Omega(\log(B))$. Since changing component only can happen at most $O(\log(N))$ times, this means, that the charging of $O(1)$ at most happens $O(\log(N)/\log(B)) = O(\log_B(N))$ times.

□

Part III
Implementation

12 Introduction

For this thesis the programming language C++ have been used[27]. The primary reason is that the language allows high level code while at the same time makes it possibly to do low level code with a minimum of overhead. C++ is designed to have an optimal run-time efficiency, so its standard library, STL[28], uses techniques such as red-black search trees to achieve best performance possibly.

This part is divided into three sections. The first describes the implementation of the trie structure used to experiment against the cache oblivious string dictionary structure. The second section describes how the cache oblivious string dictionary layout is achieved and how it is stored into a file. The last section describes the implementation of the cache oblivious string dictionary search algorithm.

Only the cache oblivious string dictionary search algorithm from [Brodal and Fagerberg, 2006] has been implemented. Therefore, the construction of the cache oblivious string dictionary layout is done in a simple and non-complex manner. Even though the alphabet is assumed to be infinitely large in [Brodal and Fagerberg, 2006], it is restricted in the implementation to be of 255 characters, just enough to be stored in a `char`. This is not exploited in any way, by for instance storing children in arrays of size 255.

13 Trie

The implementation of the trie is simple. Every allocation of a new trie node is done by using the keyword `new`, allowing C++ to put the structure anywhere it pleases (in memory). No part of the trie structure have been optimised in any way, in order to keep the trie as simple as possible. The trie node class is shown in figure 36.

```
class TrieNode{
  ChildTree *children;
  char label;
  bool isend; // is-end -> Is an endnode for a string
}
```

Figure 36: The trie node class showing the variables contained in a trie node.

[27] More precise `g++ (GCC) 4.1.1 20060525 (Red Hat 4.1.1-1)`. The program have been compiled with the parameters: `-pedantic -Wall -O3 -g`

[28] Standard Template Library.

13.1 Child tree

Each node has a pointer to a `ChildTree` structure containing the children of the node. The child tree structure can be build in different ways. For instance by using a red-black search tree or a vector[29]. The pointer makes it possible to change the child tree. Even though the alphabet is restricted in size, a search tree is used for storing the children, maintaining the desired complexity of $\log(n)$. In this thesis the `std::set` from STL is used, since it is a red-black search tree. The child tree is also allocated using the `new` keyword.

```
class ChildTree{
  void insert(TrieNode *n);
  TrieNode *search(char c);
  int size();
}
```

Figure 37: The child tree structure for the trie structure showing the functions.

13.2 Insertion and searching

Insertion is done top down, checking the child tree at each node. Therefore, the complexity for both insertion and searching in the trie structure is

$$O(|s_i| \cdot \log(n))$$

where $|s_i|$ is the length of the inserted string or the search string, n is the number of children. Therefore, searching for or inserting k strings is

$$O(|N| \cdot \log(n)), \qquad N = \sum_{i=1}^{k} |s_i|$$

[29] Also know as an array.

14 Cache oblivious layout

This section describes the implementation of the construction of the cache oblivious string dictionary structure. The construction algorithm works in steps, each adding new structure to the current, for finally being able to do a layout.

14.1 Trie

The trie is the first structure to be constructed. It resemble the trie structure from Section 13, but has a lot of functionality added, such as pointers to the blind tries and giraffe trees, component id, layer number and number of leaves beneath it, Figure 38.

```
class TrieNode{
  vector<TrieNode *> internalchildren, externalchildren;
  BridgeNode *externalsearchtree;
  GiraffeNode *giraffenode;
  BlindTrieNode *blindtrienode;
  char label;
  int componentid, layer, rank, nv, depth, cnv;
  bool isend;
}
```

Figure 38: The structure for the trie nodes in the cache oblivious string dictionary structure.

After constructing the trie, all nodes are updated concerning their depth, number of leaves and rank. It is done by traversing the trie top-down in a recursive manner. Afterwards the components can be identified. This is done top-down identifying one component at a time using the candidate requirement. If a node fails to be a candidate in the current component, it is push onto a stack of failed candidates. These are to become a roots in new components later. When all nodes for the current component have been found, the next node on the stack is selected and the identification of a new component can begin.

When all components have been identified, it is time to divide the children at each node into internal and external children which is done top-down. When constructing the trie, the children of a node are kept in a vector. When the children are divided they are places in two vectors. One for internal children and one for external. These two vectors are sorted. All children contained in another component than their parent, are placed in the external children vector.

After dividing all the children into two vectors, a weight balanced search tree is constructed using the external children. This search tree is later used when creating the component tree, and a pointer to the search tree are kept at the node. Two different search trees have been implemented. One using the algorithm from [Brodal and Fagerberg, 2006] and another using the leaf oriented optimal binary search tree algorithm.

At the same time as the children are divided a blind trie node is constructed at each component root. The reason is that when constructing the giraffe trees, the leaves must be able to refer to the next blind trie to be traversed in a search path. Having created the root of each blind trie, the leaves can do this. The blind tries are later fully created. The blind trie roots are also used in the component tree construction.

Finally the trie is traversed once again, updating the `cnv` variable. This variable indicates how many leaves a node has inside the current layer. The variable is used when creating the giraffe trees.

14.2 Component tree

Once all components have been identified, and the weight balanced search trees created, the component tree can be constructed. Again it is a top-down traversal of the trie structure. For each component the root is identified together with the bridge nodes for the component. A weight balanced tree is created using the bridge nodes with the root as the top node. To do this either the Huffman algorithm or weight balanced tree algorithm from [Brodal and Fagerberg, 2006] is used.

For constructing the component tree, three classes are implemented. In Figure 39 only the relevant details are shown for these three classes. The two subclasses `VEBBridgeNode` and `VEBComponentNode`[30] inherits from the `VEBNode` class.

A `VEBComponentNode` is used for each node in the weight balanced tree inside the component. Only the node at the top refers to the blind trie at the component root. The rest are dummy nodes discarded when doing the layout. The `VEBBridgeNode` class are used for the weight balanced search tree, connecting the components. In this way a binary component tree is constructed.

[30] The VEB refers to van Emde Boas.

```
class VEBNode{
  VEBNode *left, *right;
}

class VEBComponentNode : public VEBNode{
  BlindTrieNode *blindtrienode;
}

class VEBBridgeNode : public VEBNode{
  BridgeNode *bridgenode;
}
```

Figure 39: The nodes used to create the component tree.

14.3 Giraffe trees

The giraffe trees are constructed traversing the trie top-down. At each component, the giraffe trees are created in a depth first manner, switching between layers as the component are traversed. The giraffe trees are created using the greedy algorithm. Figure 40 shows the giraffe node class. As all information are kept inside each trie nodes, assigning the variables inside the giraffe node is easy.

```
class GiraffeNode{
  vector<GiraffeNode *> children;
  BridgeNode *externalsearchtree;
  BlindTrieNode *componentroot;
  char label;
  bool isend;
}
```

Figure 40: The giraffe node class.

Each trie node is covered by at least one giraffe tree. The giraffe tree pointer inside a trie node is referring to one of these. The pointer is set when the giraffe trees are constructed. Since the giraffe tree pointer is pointing at a giraffe tree when the blind tries are constructed, it is possible for the blind trie nodes to refer to the right giraffe tree.

14.4 Blind trie

The last structure to add is the blind trie. Again it is a top-down traversal, doing a depth first search in one component at a time. As before the trie

nodes keeps all needed information for the blind trie nodes. Figure 41 shows the blind trie node class.

```
class BlindTrieNode{
  vector<BlindTrieNode *> children;
  GiraffeNode *giraffetree;
  char label;
  int labelskips;
}
```

Figure 41: The blind trie node class.

14.5 Cache oblivious layout

The layout of the cache oblivious string dictionary structure is done in two passes. The first is a pseudo layout used to calculate the address of each node, when laid out in memory. Assuming the first node is laid out at address 0, the rest are assigned an address in the order they are traversed, following the van Emde Boas layout algorithm on the component tree.

The second pass writes the layout to a file on the disk. The output is kept in ASCII format making it possible for a human to read the final layout. The first line in the layout file is the number of bytes used. Each node is written on one line starting with an node id followed by the variables needed by the node. When writing a pointer to the file, the address of the node it points to is written. This makes it possible for the structure to be recreated by the search algorithm when loaded from a file. It is also the reason for the first pass.

To keep track of when a layout of the different layers should be done, the `std::queue` is used. In each recursive call in the van Emde Boas algorithm, a queue is given as argument. When returning from the recursive call, this queue contains the next layers to be laid out.

Using Figure 27 as example, when a recursive call of depth 2 is called, it is given a queue Q as argument. In the recursive call of depth 2 a new queue Q' is created and given as argument to the recursive calls of depth 1. When all recursive calls of depth 1 within the recursive call of depth 2 have returned, Q' contains all layer 2 of each recursive call of depth 1. Layer 0 and 1 have been laid out in the recursive calls of depth 0 and 1. The layers in Q' are now laid out, and all layer 3 are added to Q.

When doing a layout of a layer, a layout of the blind tries contained inside the layer and a layout of the associated giraffe trees is done. Both the blind tries and giraffe trees are laid out in BFS order. This means that the children

of a node is placed next to each other, and a search through these are just a matter of scanning from one end to the other

14.6 Time and space usage

The trie structure are traversed eight times to create the cache oblivious string dictionary structure, and the cache oblivious string dictionary structure twice during the layout phase. The nodes of the trie structure are sorted twice. The first time is when creating the weight balanced search trees connecting the components and the second time when creating the weight balanced trees inside each component.

If the input is n strings and $N = \sum_n |s_i|$, then the time used for creating the cache oblivious string dictionary structure and doing a layout of it is $O(\text{sort}(N))$ using the weight balanced search tree algorithm from [Brodal and Fagerberg, 2006] to connect the components. If instead the leaf oriented optimal binary search tree algorithm is used, then constriction time is $O(N^3)$.

The space usage is $O(N)$ but with a very large constant in front, as each node is represented in the trie, possibly in a blind trie, one or more giraffe trees and possibly in the component tree.

15 Cache oblivious search

The search algorithm is implemented as described in [Brodal and Fagerberg, 2006]. Unlike the construction of the cache oblivious layout, the search is cache oblivious.

15.1 Loading the CO layout

The cache oblivious layout is loaded from a layout file. The first line contains a number indicating how many bytes is used in the layout. An equivalent amount of bytes are allocated in memory before proceeding to load the nodes.

The nodes are loaded one by one and placed in the allocated memory in the order they are read. When a node is read, it is first identified by its id. Then a structure matching the node id is created, and its variables are read from the file. As the different structures are laid out in BFS order, they are read and placed in memory in BFS order. Figure 42 show the structures used in the cache oblivious layout.

```
struct st_blindtrienode { // ID 1
  char label;
  int labelskips;
  st_giraffenode *giraffe;
  int no_of_children;
  st_blindtrienode *children;
}

struct st_giraffenode { // ID 2
  char label;
  bool stringend;
  st_bridgenode *bridge;
  st_blindtrienode *blindtrie;
  int no_of_children;
  st_giraffenode *children;
}

struct st_bridgenode { // ID 3
  char label;
  st_bridgenode *left;
  st_bridgenode *right;
  st_blindtrienode *blindtrie;
}
```

Figure 42: The structures laid out in memory.

The pointers in the structures are handled different than the other vari-

ables. Instead of reading a pointer from the file an address is read. The address indicates where in memory the structure of the pointer is located, if the allocated memory started at address 0. This means that the start address of the allocated memory is added to the address read in order for the pointer to point at the right structure. The structure may or may not have been created yet making it vital that all structures/nodes are read before a search is started.

15.2 Searching in the CO layout

Searching in the structure is done as explained in Section 10. When all nodes have been read, all pointers point to the right structure, and searching is a matter of checking labels, scanning in children and following pointers.

Part IV
Experiments

16 Hardware and software

The experiments are run on three different computers, the **Internal** computer, the **Swap** computer and the **Cache** computer. The computer name refers to the experiment type that is run on the computer. The hardware specification is listed in figure 1.

Name	Description

Internal	`Intel Xeon 3,0 GHz processor`
	`16 KB level 1 cache`
	`2 MB level 2 cache`
	`800 MHz FSB`
	`2 modules of 512 MB Single Rank DDR2 RAM`
	`1 x 80 GB SATA 7200rpm harddisk.`
	The computer is running Fedora Core 5 SMP - version 2.6.18-1.2200.

Swap	`Intel Xeon 3,0 GHz processor`
	`16 KB level 1 cache`
	`2 MB level 2 cache`
	`800 MHz FSB`
	`2 modules of 512 MB Single Rank DDR2 RAM`
	`1 x 80 GB SATA 7200rpm harddisk.`
	The computer is running Fedora Core 5 SMP - version 2.6.18-1.2200, booted with 80 MB RAM.

Cache	`Intel Pentium 4 processor 3,4 GHz`
	`16 KB level 1 cache`
	`1 MB level 2 cache`
	`800MHz FSB`
	`2 modules of 512 MB DDR2 400 NECC Dual Channel RAM`
	`3 x 400 GB SATA 7200 rpm harddisk.`
	The computer is running Fedora Core 3 SMP - version 2.6.12-1.1381 booted with PAPI included in the kernel.

Table 1: Hardware specifications for each computer used in the experiments. The computer name refers to the kind of experiments that are tested on the computer. On the **Internal** computer the experiments are run in internal memory only. The **Swap** computer is used for experiments where swapping to external memory is required and the **Cache** computer are used to count the number of cache misses.

A few software programs have been used in connection with the experiments. Some of these are used as a part of the experiments, and some for

analysing the result of the experiments. These programs are shortly described in the following subsections.

16.1 PAPI

The `PAPI` [Dongarra *et al.*, 2003] software makes it possible to count the number of cache misses in cache level 1 and 2 on the CPU. A modified linux kernel including the `PAPI` library is needed to enable the use of the `PAPI` library in `C++`.

The level 1 cache is usually divided into two parts. One is holding the program instructions and the other the data to be processed. This makes it possible to count the data cache misses and the instruction cache misses separately in the level 1 cache. The level 2 cache is not divided and therefore, it is not possible to tell the different kinds of cache misses from each other. Only the number of total cache misses are available.

16.2 Perl

In order to perform the experiments in succession and to minimise user interaction, the experiments are executed by programs written in `Perl` [Wall, 2006]. As this is the case for all experiments the small amount of memory used by the `Perl` interpreter, is similar for all experiments.

16.3 Gnuplot

The graphs used to analyse the data are all made in `Gnuplot` [Williams and Kelley, 2004], from the raw data output of the experiments.

17 Data sets

To get useful information from the experiments, a large amount of data have been generated. The data are generated to have specific properties that are interesting in the context of the experiments. The cache oblivious string dictionary structure have also been tested with real life data, as performance measurement on synthetic data not necessarily behave the same way. Every synthetic experiment has been generated in 5 variations.

All data for the experiments are denoted by a letter and a number. This is done to be able to distinguish them from each other. When referring to data by a single letter, the whole data set with this letter is referred to. For instance the data set A includes the data elements A1, A2, A3, A4 and A5.

The data sets A to D are all synthetic generated, while E and F are made from real life data. The synthetic data are generated to give certain properties to the trie structure that are formed when the data is inserted. These properties are interesting when the components, blind tries and giraffe trees are build.

For each individual set of strings generated for a data set, a similar set is generated. In this similar set a character in each string is replaced by another character, which is not in the original alphabet. This is done to be able to run experiments, where the strings do not match, but still completes part of a search. The character to be replaced is chosen randomly.

17.1 Data set A: Long strings with few splits

The strings in this set are all of the same length. They are generated so that the paths in the trie structures consists of many unary nodes. Each path splits ten times. A path splits into at least two and at most three paths. Figure 43 (a) shows an example off such a trie and Table 2 shows the variations of the number of strings and their length.

Since the paths in the trie structure consists of many unary nodes, the rank of the nodes does not increase very often. This will result in fairly large components. The long paths without splits will result in small blind tries, but long giraffe trees.

The data is generated bottom up using the parameters in Table 2. The construction algorithm starts by creating the last 9% of all strings. These 9% are all different from each other. Then for each three strings the paths are merged. This is done by letting the three strings share the characters in the next 9% of their length.

As the final result should be a tree with ten splits on a root to leaf path, mergin three strings at a time may be too much. Therefore, when the number

of individual strings reaches the amount needed to merge only two strings at a time this will happen.

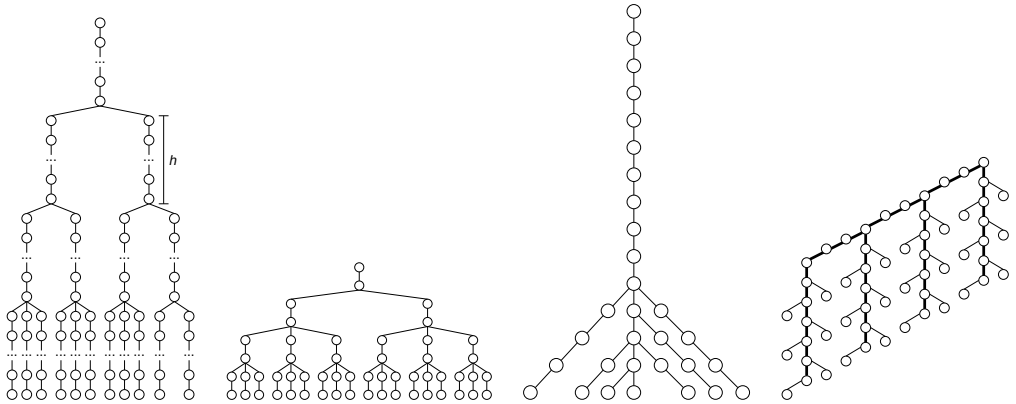


Figure 43: Example structures of the data sets A, B, C and D. An example of data set A is illustrated in (a), an example of data set B in (b), an example of data set C in (c) and an example of data set D in (d).

Name	Number of strings	String length
A1	5000	4400
A2	7500	3200
A3	10000	2275
A4	12500	1825
A5	15000	1275

Table 2: Properties of data set A

17.2 Data set B: Short string with many splits

This data set is similar to A. The only difference is that the paths in the trie structure is very short. Again the strings are all of the same length, but with few unary nodes. The variations of the data is shown in Table 3 and in Figure 43 (b) an example of the trie structure is given.

The resulting trie will have many components as the rank often shifts. The resulting blind tries will almost be identically to the original trie.

The data is generated top down, by adding prefixes to the already existing strings. Every time three diffent prefixes is added to an already existing

Name	Number of strings	Number of nodes before split
B1	1250000	0
B2	750000	1
B3	500000	2
B4	425000	3
B5	375000	4

Table 3: Properties of data set B

Name	Number of strings	String length	Max strings with same prefix
C1	15000	1500	5000
C2	15000	1500	7500
C3	15000	1500	10000
C4	15000	1500	12500
C5	15000	1500	15000

Table 4: Properties of data set C

string. This is done for all existing strings before starting to add prefixes to the newly generated strings. The length of the added prefix and the number of generated strings are varied.

When the required number of strings is generated, the remaining strings (if any) strings are padded so that the number of strings remains the same and the length is the same for all strings.

17.3 Data set C: Long strings with many splits at the end

Strings in this set have more than 70 % of their prefix in common. The trie structure will consist of one or more long path of unary nodes for the first 70%. The last 30% is the bottom, where each node can have several children. Figure 43 (c) show an example of this. All the strings in this structure has the same length. The variations of the data is shown in Table 4.

The data are generated to test behaviour on very large components. The entire trie is generated with a long path of unary nodes before any split. Therefore, the rank does not change in the topmost strata. This results in one big component at the top and several small in the bottom. The giraffe trees for this data set should be very large because of the long unary path.

Name	Number of strings	Number of initial strings
D1	130000	20
D2	130000	30
D3	130000	40
D4	130000	50
D5	130000	60

Table 5: Properties of data set D

When generating the strings in this set, a single string is created.

The generation of strings is started by generating a single string. The remaining strings are generated with a random percentage between 70% and 95% of the first string as prefix. All strings have the same length as the first.

17.4 Data set D: Long strings with many splits

This set consists of two subsets, a *parent* set and a *children* set. The parent set consists of long strings sharing only some of their prefix. In Figure 43 (d) this is the long straight highlighted lines.

The elements in the child set are build from elements from the parent set. Each element in the child set shares all but one character of one of the elements in the parent set. The child elements are those ending in one node branching from the highlighted lines in Figure 43 (d). The variations of the data is shown in Table 5.

The data results in components with increasing size, as the number of children decreases by one every time the depth increases by one. The blind tries is almost similar to the original trie, as the original trie splits at every node.

The data is generated by first generating a long string. This string is split up in a number of initial strings depending on the parameter. Then for each initial string, a string is generated that has all but the last character of the initial string. The new string is then appended two new characters. This happens for all initial strings, then for the newly generated strings and so forth until the required amount of strings is reached.

17.5 Data set E: Shakespeare

The strings are made up of all the individual words from the collection of Shakespeare comedies and tragedies. The trie structure will consist of rel-

Property	Value
Number of strings	67505
Average string length	7.5
Average fan out (over all nodes)	1.55
Average fan out (where nodes split)	3.63

Table 6: Properties of data set E

atively short strings. Some nodes will have a big fan out while others only have a few. The properties are shown in Table 6.

17.6 Data set F: DNA strings

The data set consists of substrings of length 100 from a representation of a human chromosome. This data is different from the E experiment, in that chromosome data only consist of a small number of different letters, and in that repetitions are common. The properties are shown in Table 7.

Property	Value
Number of strings	51878
Average string length	100
Average fan out (over all nodes)	1.01
Average fan out (where nodes split)	2.55

Table 7: Properties of data set F

18 Experiment procedure

All the experiments have been executed by `Perl` scripts. As experiments on the naive trie structure and the cache oblivious string dictionary structure are executes by the scripts, the small amount of memory usage caused by the scripts are the same.

Each search is performed two times for each input to minimise fluctuations in the running time of the experiments. The two searches are not performed in succession to be sure none of the used data still resides in the cache after the first experiment.

The experiments includes execution on the **Internal** , the **Swap** and the **Cache** computer. All data are kept in internal memory when experimenting on the **Internal** computer. When experimenting on the **Swap** computer, some of the data are kept in swap memory. The experiments for data sets D and E are not performed on the **Swap** computer, as the data fit into internal memory. When experimenting on the **Cache** computer, the number of cache misses are counted using PAPI. All data are kept in main memory when using the **Cache** computer.

18.1 Trie structure

An experiment on the trie structure starts by loading all strings without errors[31]into memory. Then the strings are inserted into the trie structure. The trie is searched for both the set of strings without errors and the set of string with errors. This is done independent of each other. The construction of the trie structure and the searching for each set is timed.

18.2 Cache oblivious string dictionary

An experiment on the cache oblivious string dictionary structure is started by first constructing a layout of the corresponding trie structure[32]. This is done by one program. The trie structure is generated as described in Section 18.1. Then the cache oblivious string dictionary structure is created and laid out in a file.

Another program reads this file together with the set of strings to be examined. When both the layout and all the strings are contained in memory, the search is started. Again both the set of string without errors and the set with errors are tested.

[31]The set of string, where no character has been replaced, Section 17.

[32]If such a layout already exists, this step is skipped.

The construction of the layout file and the associated loading of the layout is timed. So are the searches in the loaded structure.

The layout files are constructed on different computers. This is done as the cache oblivious string dictionary structure takes up a lot of memory[33], and because the total size of all layout files exceeds the maximum available space on one computer. The script files take this in account when running the experiments.

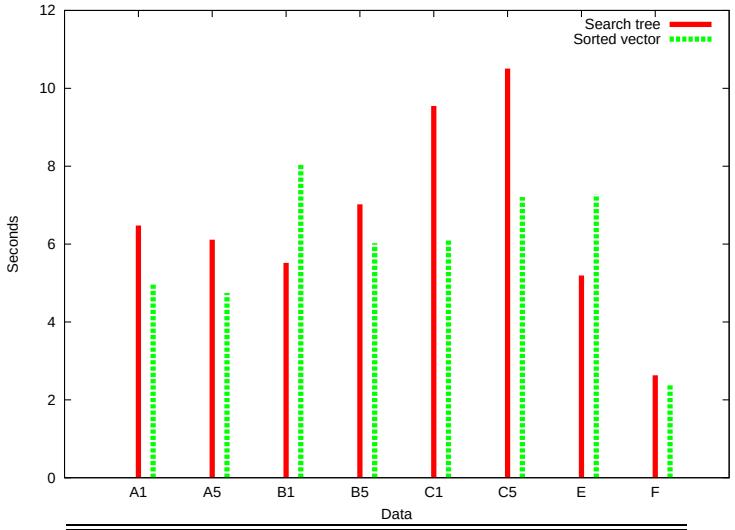
18.3 Cache misses changes

When experimenting on the **Cache** computer, a modified version of the program searching in the cache oblivious string dictionary is used. It has been modified to use the PAPI library so it can count the cache misses in level 1 cache level 2 cache. Apart from the counting functions the program is the same.

[33]See Section 14.6.

19 Trie experiments

The trie experiments consist of testing the naive trie structure on all data sets on all three computers[34]. This is done to analyse the behaviour in internal memory, swap and to look at the generated cache misses.



Data set	Childtree	Execution time (in seconds)
D1	Search tree	799.1
D1	Sorted vector	558.8
D5	Search tree	319.7
D5	Sorted vector	249.9

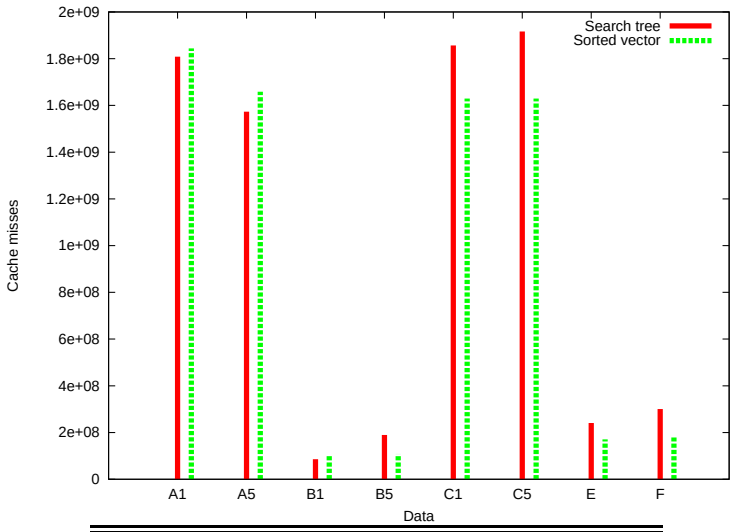
Figure 44: Execution time for the trie structure on the data sets A to F (D in table) in internal memory. Only the two extreme elements from each set is shown.

As describes in Section 13.1, two different child trees are implemented. One using a vector and one using a red-black search tree. As both have the same theoretical search time, $O(\log(n))$, it is expected that the results will be similar. This is however not the case when looking at the results in figure 44. The figure shows the results of the experiments done in internal memory. It shows that in most cases it is faster to use the sorted vector.

However, this might be due to implementation specific details in the `std::set`. The nodes in the red-black search tree might be places arbitrary in memory, while a sorted vector is placed in succession. In this way a

[34]The **Internal** , **Swap** and **Cache** computer.

search in a vector will result in fewer cache misses than the red-back search tree. This theory is backed up by the graph in Figure 45.



Data set	Childtree	Cache misses
D1	Search tree	$6.54 * 10^{10}$
D1	Sorted vector	$4.04 * 10^{10}$
D5	Search tree	$3.43 * 10^{10}$
D5	Sorted vector	$2.58 * 10^{10}$

Figure 45: Level 1 cache misses for the trie structure using either a sorted vector or a red-black tree.

It is not much that can be concluded from the results of the swap experiments. One reason is that it is not known how the trie structure is laid out in memory. The difference in execution times in figure 46 is possibly a result of an alignment fitting for the block size. It can be concluded that storing the children in a vector is superior to storing the children in a red-black search tree, when experimenting on the data set C. The execution time is also much higher when swapping as expected.

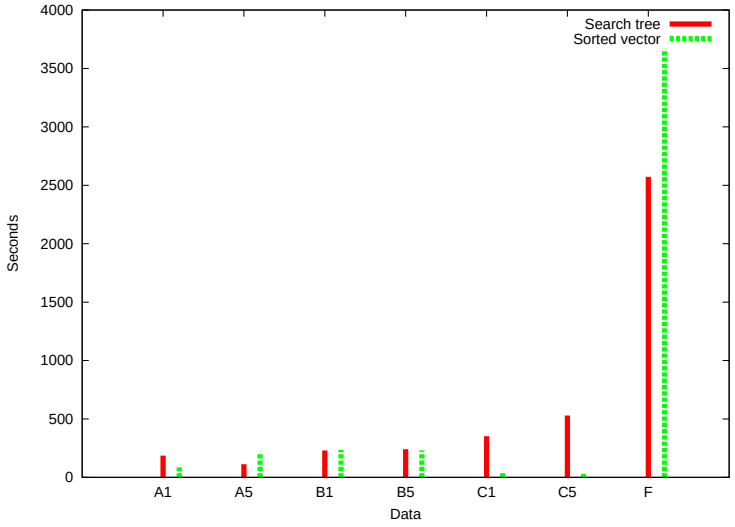


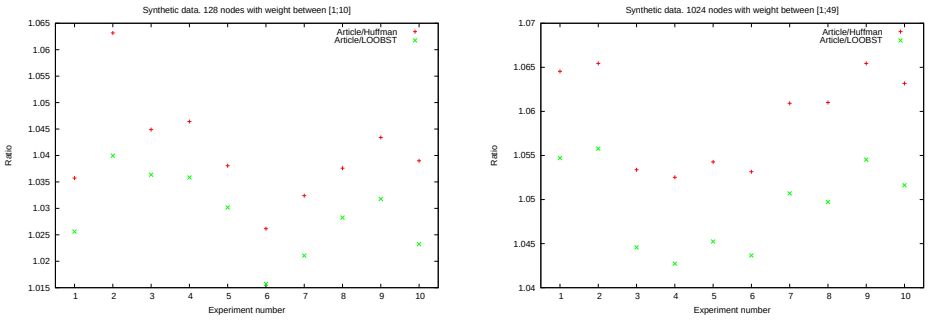
Figure 46: Execution time for the trie structure on the data sets A, B, C and F when swapping. Only the two extreme elements from each set is shown. Each data element uses approximately 120 MB of memory.

20 Weight balanced trees

In [Brodal and Fagerberg, 2006] a weight balanced search tree is described. The construction algorithm takes a list of sorted key with weights as input and creates a weight balanced tree[35]. A leaf in this tree with key x_i and weight w_i is at most at depth $2 + 2\lceil \log(W/w_i) \rceil$, where W is the sum of the weights of the keys. This tree can be constructed cache oblivious in $O(n)$ time using $O(n/B)$ I/Os. Since the construction algorithm in this thesis is not cache oblivious, other trees can be used instead.

20.1 Other weight balanced trees

The article tree is used in two different constructions in the cache oblivious string dictionary. It is used as a weight balanced tree inside components and as a weight balanced search tree connecting the components. To replace the weight balanced tree the Huffman tree is chosen and to replace the weight balanced search tree the leaf oriented optimal binary search tree[36] is chosen.



(a) (b)

Figure 47: The ratio between the Huffman tree, the article tree and the leaf oriented optimal binary search tree. In (a) 128 leaves are used each with weights between 1 and 9 uniformly distributed. In (b) 1024 leaves are used with weights between 1 and 49 uniformly distributed. A total of 10 trees have been constructed.

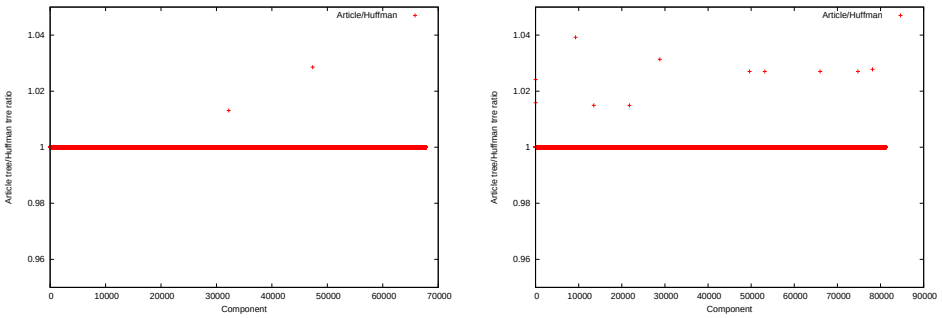
Figure 47 shows the ratio between these search trees and the article tree. The ratio is found by dividing the total weight of the article tree with the total weight[37] of either the Huffman tree or leaf oriented optimal binary search tree. In Figure 47 (a) a tree is constructed using as input 128 nodes

[35] Denoted as *the article tree*.
[36] LOOBST for short.
[37] See Section 5 for a definition of the total weight.

with a weight $\in [1;9]$. In (b) 1024 nodes are used all with a weight $\in [1;49]$. The weights are uniformly distributed. It is clear, that in both cases the Huffman tree and the leaf oriented optimal binary search tree is superior to the article tree as all values is above 1.

20.2 Huffman tree vs. Article tree

Although the Huffman tree is superior to the article tree on uniformly distributed data, this is not necessarily the case with the experiment data, i.e. the data sets A to F. As the fan out is at most three in the data sets A to D, only the data sets E and F are interesting.



(a) (b)

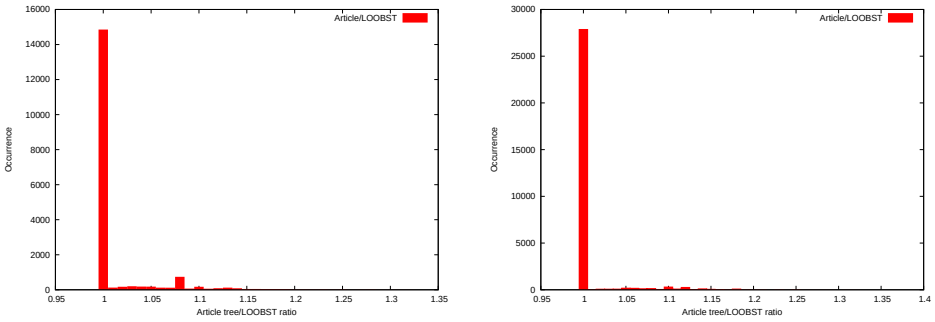
Figure 48: The ratio of the total weights in the Huffman tree and the article tree when constructing the weight balanced tree inside components in data sets E (a) and F (b).

In Figure 48 the ratio is shown for the Huffman tree and the article tree for the data sets E and F. The total weight for each weight balanced tree in each component is measured. It shows that only in a very few cases the Huffman tree performs better. In most of the cases the weights are the same. The difference between using the Huffman tree and the article tree is negligible.

20.3 Leaf oriented optimal binary search tree vs. Article tree

As with the Huffman tree, the ratio between the article tree and the leaf oriented optimal binary search tree for the data sets E and F is measured. Figure 49 shows the ratio occurrences. The ratio is measured for each bridge tree, i.e. the trees connecting the components.

The results differ slightly from the Huffman results. In some cases the leaf oriented optimal binary search tree is better than the article tree. The ratio indicates though that the difference in not much.



(a) (b)

Figure 49: The ratio of the total weights in the leaf oriented optimal binary search tree and the article tree when constructing the search tree between components for data sets E (a) and F (b).

20.4 Concrete example

To see if there are any gain in execution time when using either the Huffman tree or the leaf oriented optimal binary search tree experiments on the data sets A to F have been run. The results[38]. for the Huffman experiment is shown in Figure 50

The columns in Figure 50 (a) shows that the execution time is almost identical. However, the results for the swap experiments does show a difference. The Huffman tree is not always better, but for the majority of the tests it is.

The same is done the leaf oriented optimal binary search tree. The results are shown in Figure 51. The tendency is the same, when using the **Internal** computer but opposite when using the **Swap** computer.

[38]The results from the data set D is omitted as the column would be to high to fit into the diagram.

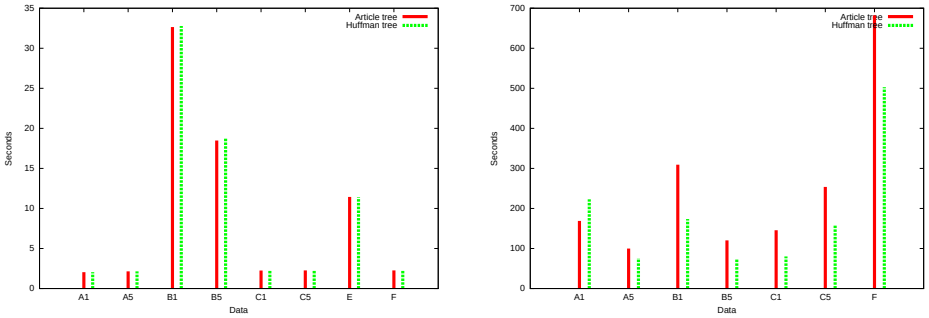


Figure 50: Execution time for the cache oblivious string dictionary when using the Huffman tree instead of the article tree inside components. In (a) the **Internal** computer is used and in (b) the **Swap** computer.

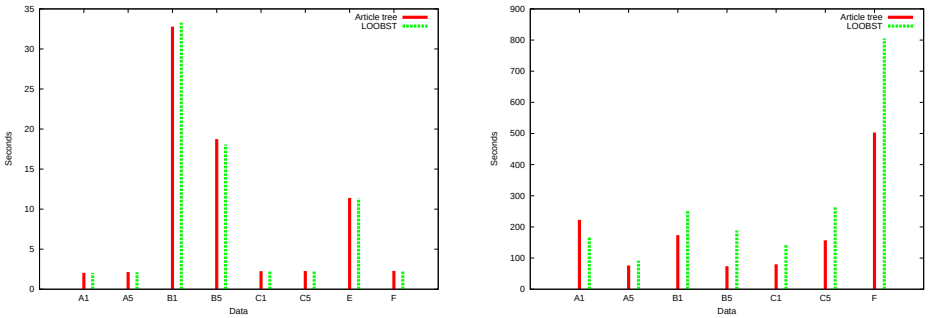


Figure 51: Execution time for the cache oblivious string dictionary when using the leaf oriented optimal binary search tree instead of the article tree inside components. In (a) the **Internal** computer is used and in (b) the **Swap** computer.

21 ε experiments

When constructing the cache oblivious string dictionary from a data set, the components generated depend on which strata the nodes from the original trie is placed in, and the difference of rank between the nodes. The theory dictates that two nodes are in the same component as long as $\text{rank}(a) - \text{rank}(b) < \varepsilon 2^i$, where i is the strata to which they belong.

According to the theory, ε should be $< 1/2$ for a search path P to be traversed in $O(\log_B(n) + |P|/B)$ I/Os. This is not necessarily true when the input is not worst case. Therefore, the experiments in this section is run for values of $\varepsilon > 1/2$.

The experiments are run with the Huffman tree inside the components, the article tree connecting the components and giraffe trees where half of the nodes are ancestors.

21.1 Data set A

The results of the experiments on the data set A is shown in figure 52. The graph shows that the execution time is best for large values of ε. This does not correspond well with the theory, as $\varepsilon < \frac{1}{2}$ should be best. Table 8 indicates that the execution time depends heavily on the number of total bridge nodes in the layout. The overhead from searching in the bridge nodes is the course for the execution times.

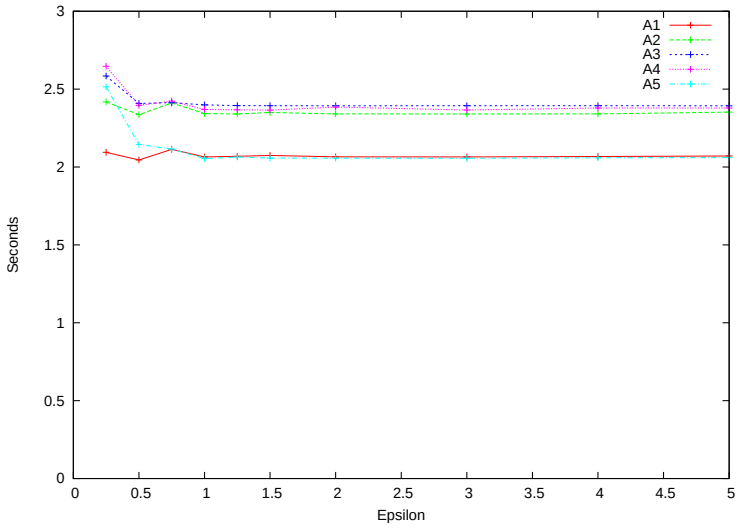


Figure 52: Execution time for the cache oblivious string dictionary structure on the data sets A for various values of ε.

The number of bridge nodes depends of how many components there are in the layout. Table 8 shows the different number of components for the A1 experiment. The number of component shifts much for small values of ε. This is because when ε is 0.25 or 0.75 the last part of each string becomes a component. The data set A is constructed so that the rank shifts at the same place for each subtree.

ε	**Bridge node size**	**Components**
0.25	142064	5427
0.5	4784	186
0.75	136160	5195
1	0	1

Table 8: Properties of the layout of A1 for varying ε.

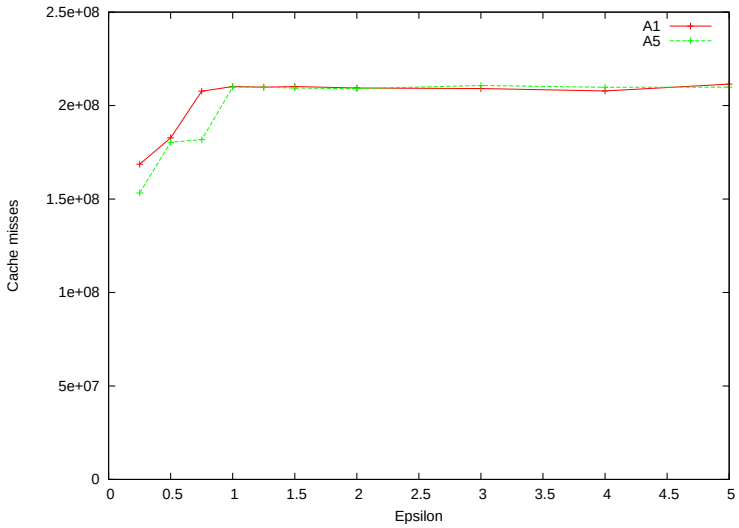


Figure 53: Level 1 data cache misses for data set A for various values of ε.

The graph in Figure 53 shows that the number of level 1 cache misses increases when ε increases. The reason for the increase is likely to be due to the number of nodes included in a layer. If a path in the trie do not split often, the corresponding giraffe tree will contain many nodes and few splits. As the number of nodes in the layers increases, the length of the giraffe trees will increase.

When the giraffe trees are short, they are able to fit into memory. Some of them may then be reused when searching for the next string. The long giraffes cannot be reused often as they typically only have a few children.

Many of the long giraffe trees will be identical along the neck, but have different children. This means that many almost identically giraffe trees are loaded into memory.

As the giraffe trees becomes very long, the execution time is still quite good even though the number of cache misses is high. This can be seen by looking at the execution time for experiment A5 and the number of level 1 cache misses. The reason is that modern CPUs uses prefetching, meaning they do some read ahead. When a block of data have been processed, the next block have already been fetched. It might be that the fetched block is not the next to be processed, but in most cases it is. Even though, fetching the block counts as a cache miss.

21.2 Data set B

The experiments on data set B behaves somewhat similar to the experiments on data set A. For large ε the execution time gets better. There are also some spikes in the behaviour of the of the graph for data B1.

It can be seen in Figure 54 that the graphs behaves the same for ε up til 1. This is because the data are constructed with a large number of paths that splits often. Each node has 3 children so the difference in ranks will be either 1 or 2 from node to node.

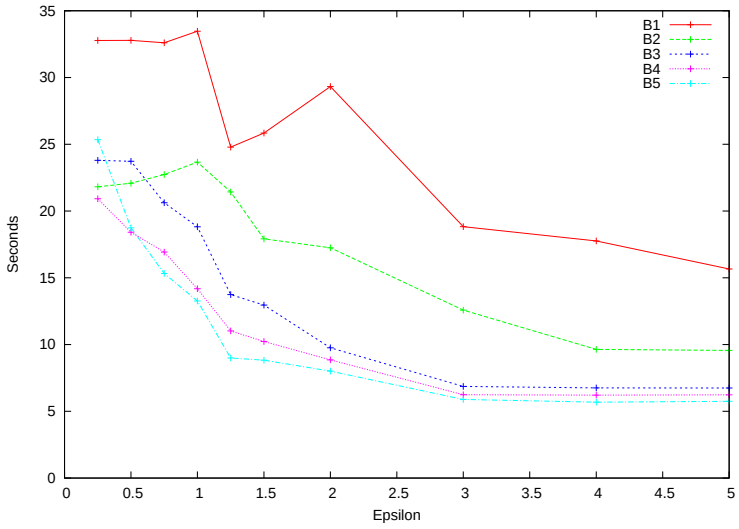


Figure 54: Execution time for the cache oblivious string dictionary structure on the data sets B for various values of ε.

Because of the rank difference between each node, every node in B1 will create a new component when $\varepsilon \leq 1$. This has a drastic effect on the execu-

tion time. Changing from one node to a child node means going through a blind trie node, a giraffe tree node and a bridge of up to three nodes.

When ε becomes 1.25 the nodes where the rank difference is only 1 will become one component. This means the total number of components will drop. As all strings are searched for, the total number of traversed bridge nodes will also drop making the execution time smaller.

Looking at the graph for B1 in Figure 54, there is a increase for $\varepsilon = 2$. This is because the number of one-node components[39] increases from when ε was 1.25 (and 1.5). When the number of one-node components increases, then the number of bridge node traversed in a search path increases. Figure 55 shows this by a small example.

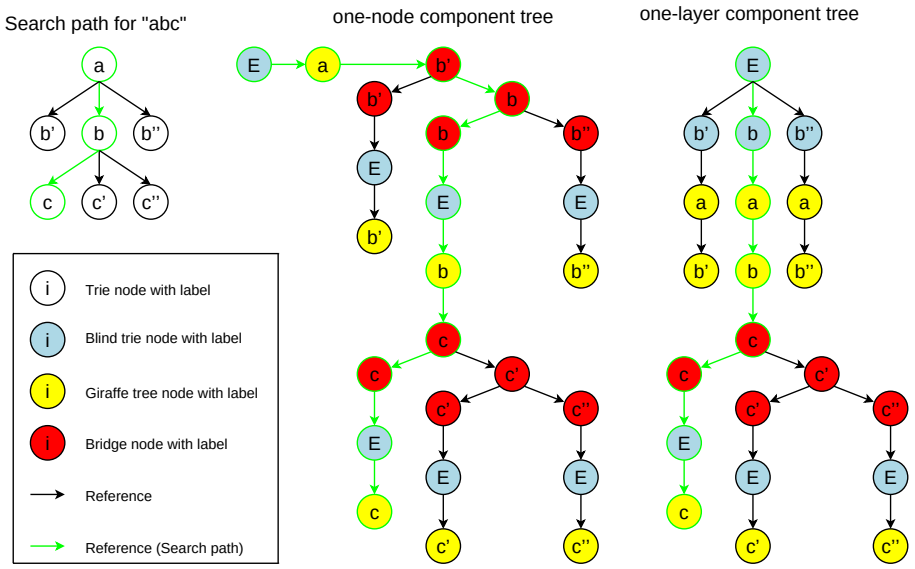


Figure 55: A search path in a trie (left), in a one-node component tree (middle) and a one-layer component tree (right).

The search path in the trie is shown to the left. The middle tree shows the same search path in a one-node component tree and the right tree shows the search path in a one-layer[40] component tree. The nodes are ordered so that $b' < b < b''$ and $c < c' < c''$. The difference between the search path in the one-node component and one-layer component is three bridge nodes.

The graph in Figure 56 shows level 1 data cache misses. It has a nice coherence with the graph in figure 54. The increased in the number of bridge

[39] Components containing only one node.
[40] Components with only one layer.

nodes also increases the number of cache misses, as less data are reusable in the next search.

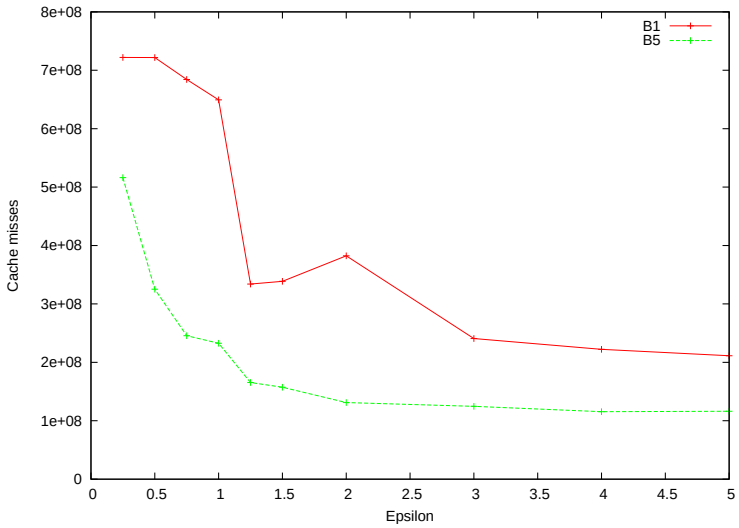


Figure 56: Level 1 data cache misses for data set B for various values of ε.

21.3 Data set C

The results for various ε on data set C show the opposite behaviour than the experiments on A and B did. Figure 57 shows the results for the experiments on C for various ε.

The steep increase in execution time for $\varepsilon = 1$ is due to the entire structure being in one component. As data set C consist of strings sharing 70 % to 95 % of their prefix, the giraffe trees will have at least 2 children each and probably around 4 in average. The exact count for data C1 for $\varepsilon = 5$ is 3348 giraffe trees. As there is 15000 strings this is a bit less than 5 children for each giraffe tree.

As each string is searched for, each fifth string requires an entire quite large giraffe tree to be loaded. For a small ε, there will be many components sharing the same giraffe tree. Since the giraffe tree will be small it can be kept in the cache. Therefore, the number of cache misses will decrease. This can be seen in figure 58. The cache misses are more than halved for small ε resulting in the decrease in time.

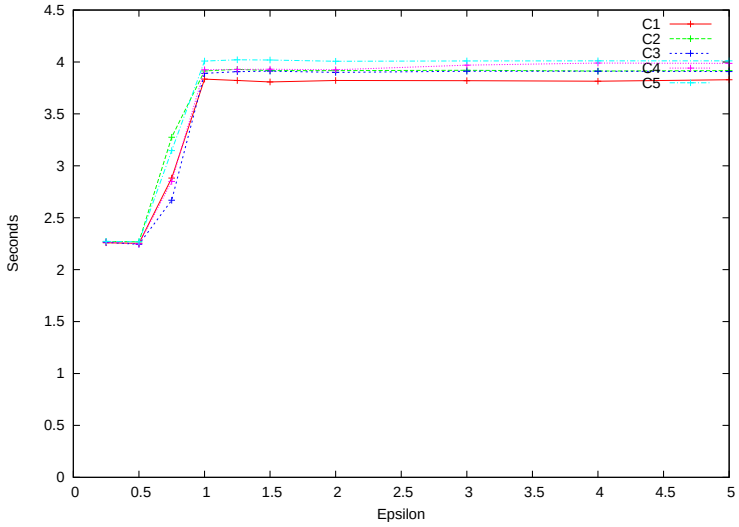


Figure 57: Execution time for the cache oblivious string dictionary structure on the data sets C for various values of ε.

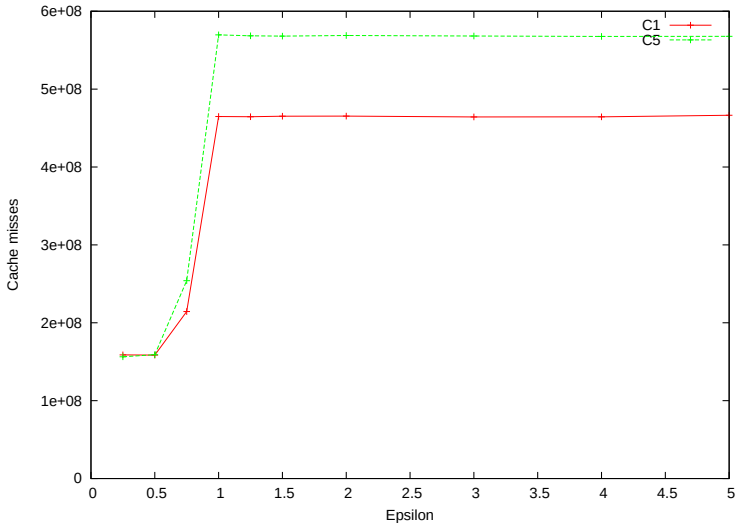


Figure 58: Level 1 data cache misses for data set C for various values of ε.

21.4 Data set D

Figure 59 shows the execution times for data set D. It resembles Figure 60. The execution time rises drastically when $\varepsilon \geq 1.25$.

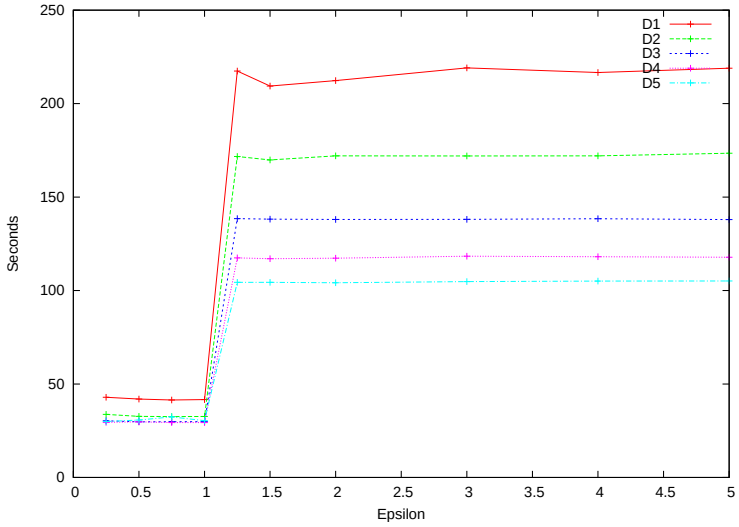


Figure 59: Execution time for the cache oblivious string dictionary structure on the data sets D for various values of ε.

The reason is very long giraffe trees, covering the same neck with only relatively few children. In data set D there is 130000 strings, which results in 130000 children in the trie. The root will then have rank 18. As can be seen in Section 17, the trie have nodes with rank one at every level down the trie. At strata 4 for $\varepsilon = 1$ the rank difference should be below $1 \cdot 2^4 = 16$. As this is not the case every end node of a string in strata 4 is put in a component.

When $\varepsilon = 1.25$ the difference should be below 20 so the single nodes in strata 4 does not need to be put in their own components. This results in a very large component covered by long giraffe trees, which will only be reused a small amount of times when searching for the strings. As seen in Figure 60 this results in many cache misses, and also in increased execution time.

21.5 Data set E

Not much is known about the layout of data E. As seen in figure 61 the execution time declines steadily, as ε rises. This is the result of fewer components for larger ε resulting in fewer bridge nodes. The number of bridge nodes in the layout decreases from 119864 to 8999. The steepest decrease

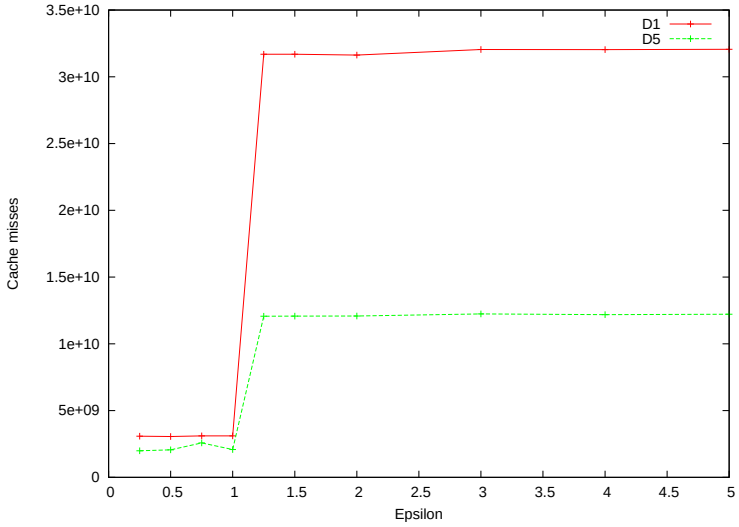


Figure 60: Level 1 data cache misses for data set D for various values of ε.

is around $\varepsilon = 1.25$ where the bridge node count decrease with almost 33 %. This can also be seen in both Figure 61 end 62.

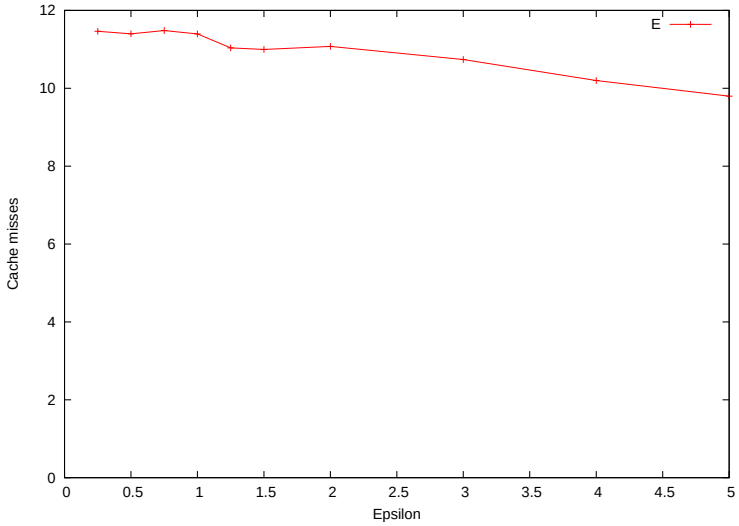


Figure 61: Execution time for the cache oblivious string dictionary structure on the data sets E for various values of ε.

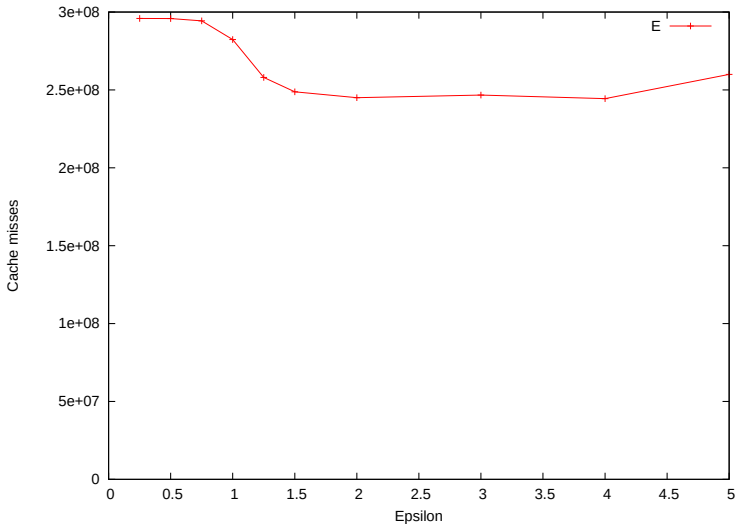


Figure 62: Level 1 data cache misses for data set E for various values of ε.

21.6 Data set F

Figure 63 shows the execution time for data F. It is approximately the same as 61. The explanation is again that the bridge node count decreases rapidly. The two string sets made from real life data behaves very similar, even though they are taken from two completely different real life situations.

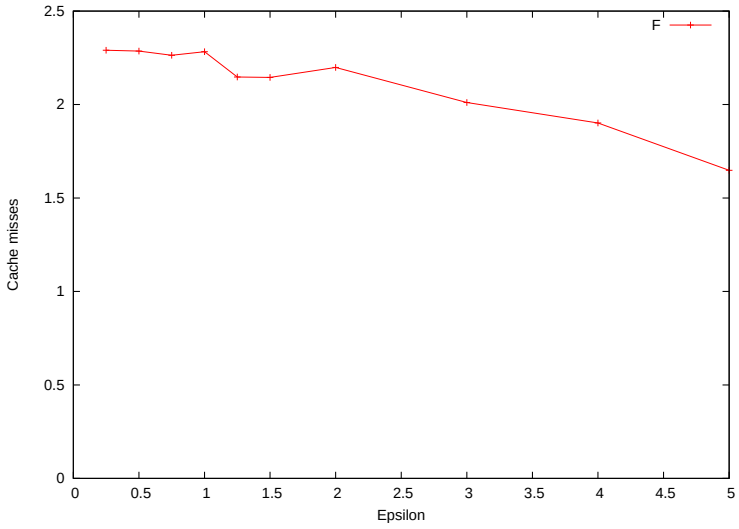


Figure 63: Execution time for the cache oblivious string dictionary structure on the data sets F for various values of ε.

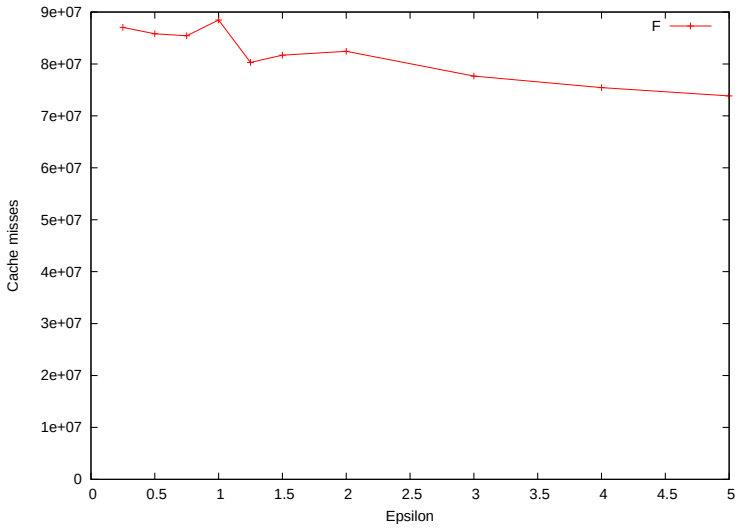


Figure 64: Level 1 data cache misses for data set F for various values of ε.

21.7 Comparison of execution times

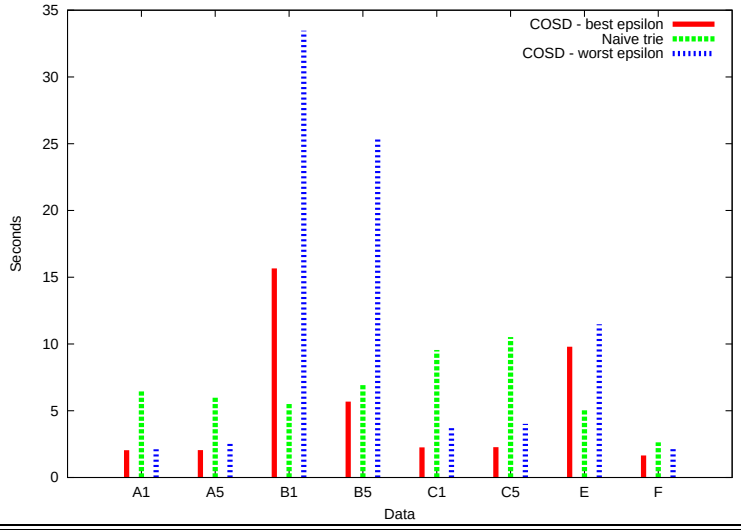
Figure 65 shows the execution time for the naive trie and the cache oblivious string dictionary. Only the best and worst ε is shown. The times indicates that searching in the cache oblivious string dictionary is faster for the right ε for most of the data sets.

The type of data where searching in the cache oblivious string dictionary are best, are data that has many unary nodes in succession. In these cases the number of cache misses can be reduced and the CPU prefetching mechanism is working perfectly.

The naive trie has better execution time when searching in the data elements B1 and B5. In the data set B there are few unary nodes are in succession and most of the components contain only one node. The overhead in traversing bridge nodes and giraffe nodes becomes an disadvantage for the cache oblivious string dictionary. It is the same case for the data set E.

When looking at the execution times in swap, the result is different. Figure 65 shows that for the right ε, searching in the cache oblivious string dictionary is better than searching in the naive trie. The right ε can vary from data set to data set. In most of the cases, choosing a poor ε results in an execution time not far from the execution time of the naive trie.

When placing the layout in swap memory, the overhead from bridges in data set B is negligible.



Data element	COSD, best	Naive trie	COSD, worst
D1	41.4	799.9	219.1
D5	29.9	319.7	105.1

Figure 65: Execution times using the cache oblivious string dictionary (best and worst ε) and the naive trie.

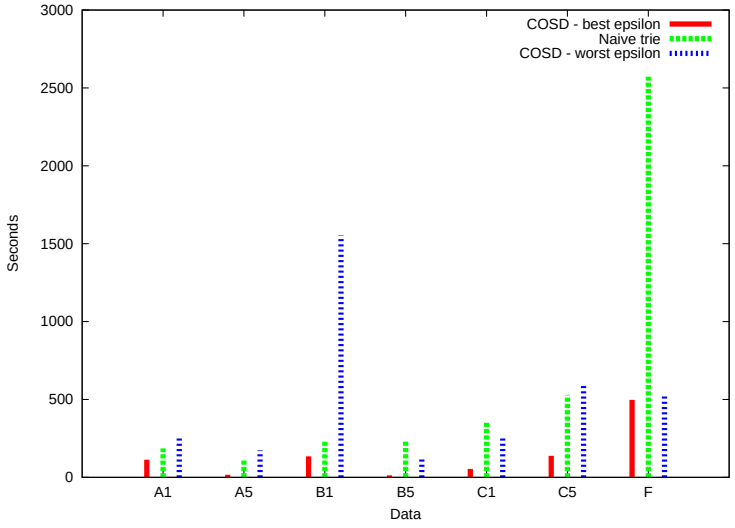


Figure 66: Execution time for searching in the cache oblivious string dictionary (best and worst ε) and the naive trie on the **Swap** computer

21.8 Conclusion

It is hard to establish an ε that is optimal. It seems that $\varepsilon < 1/2$ works well on most of the synthetic data that has a large number of succeeding unary nodes. The larger ε is better for the synthetic data set B as well as for data set E and F.

As E and F consists of real life data it can be argued that it will be best to have a large ε. However, the speedup is very small compared to the penalty the other data sets suffer from large epsilon. A relatively low ε seems to be the best solution.

The comparison between the trie and the best parameter of ε suggest that the cache oblivious string dictionary is superior for the right ε.

22 Giraffe tree experiments

An interesting experiment is to change the theoretical length of the giraffe trees. The theory states that a giraffe tree is a tree where more than $N/2$ of the nodes are ancestors to all leaves. The proof for the execution time and space usage still hold if the neck of the giraffe tree is different from $1/2$. Every constant number K in $]0,1[$ can be used.

$$\begin{aligned} |A^{i:j}| + |B^{i:j}| &> \frac{|T^{i:j+1}|}{\frac{1}{1-K}} \Rightarrow \\ |T^{i:j}| < |T^{i:j+1}| &< \frac{1}{1-K}\left(|A^{i:j}| + |B^{i:j}|\right) \Rightarrow \\ \sum_{T^{i:j}} |T^{i:j}| &< \sum_{T^{i:j}} \frac{1}{1-K}\left(|A^{i:j}| + |B^{i:j}|\right) \leq 2\frac{1}{1-K}N \end{aligned}$$

The same holds of the proof of execution time. If the string searched for is less than $N \cdot K$ and as the giraffe tree is laid in memory in BFS layout, then the search time is $O(p/B)$. Is the search string more than $N \cdot K$, then it has already matched the first $N \cdot K$ and is at most $\frac{1}{K} \cdot O(p/B) = O(p/B)$.

When the neck of the giraffe is a large percentage of the tree the space usage increases, while the search time decreases. The opposite is also true. The experiments changes the percentage of the nodes needed to be ancestors. The Huffman tree is used inside the components, while the leaf oriented optimal binary search tree is used to create the search tree connecting the components. ε is 0.5.

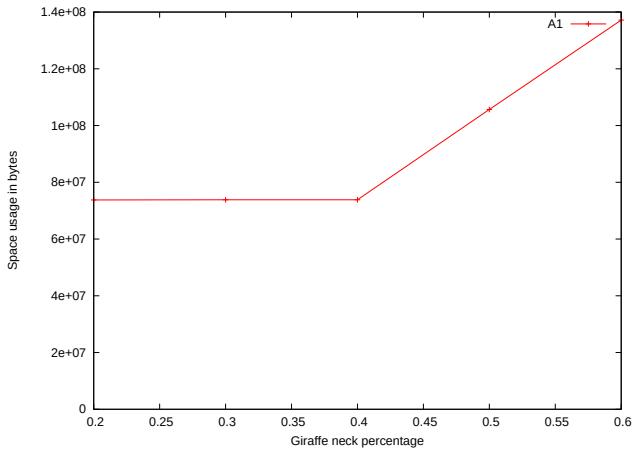


Figure 67: Giraffe tree space usage for various giraffe neck percentage of A1.

The percentage of the giraffe neck is varied between 0.2 and 0.6. A higher percentage than 0.6 is not used because the space usage increases as more giraffe trees are needed. Figure 67 illustrates the increase in space usage as the percentage increases.

22.1 Data set A

Figure 68 shows that the execution time do not seem to fluctuate by varying the neck percentage. However, it do seem that it influences the amount of cache misses, Figure 69. This is accordance with the theory as the theoretical search time should decrease, when the space usage and amount of giraffe trees increases.

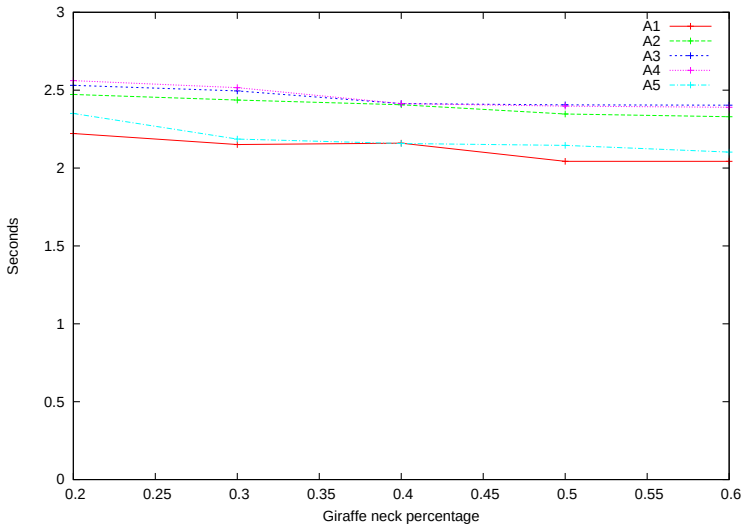


Figure 68: Execution time for the cache oblivious string dictionary structure on the data sets A for various giraffe trees.

When searching a giraffe tree, where the nodes of the neck only constitute 20 % of the giraffe, more data needs to be processed than when searching a giraffe where the neck constitutes 60 %. However, it do not seem to influence execution time. This can be due to the fact that a giraffe tree lies consequently in memory so that the CPUs prefetching mechanism will have it ready when the data is needed.

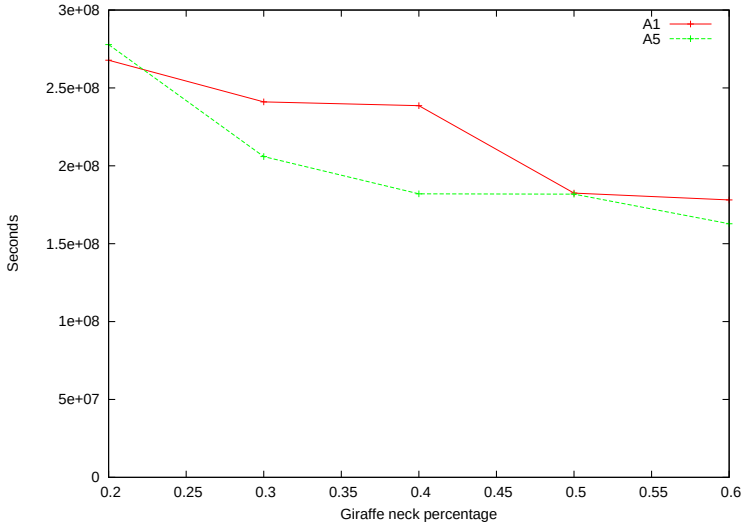


Figure 69: Level 1 data cache misses for data set A for various giraffe trees.

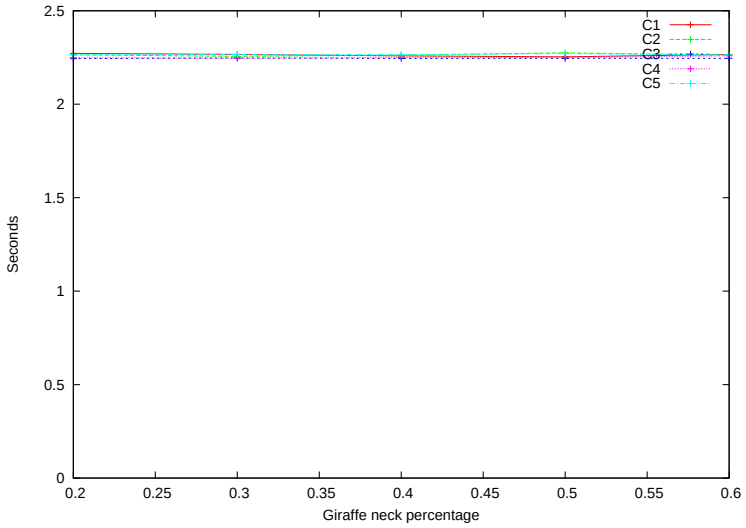


Figure 70: The graph shows the execution time for the cache oblivious string dictionary for various giraffe trees for data set C.

22.2 Data sets B, C, D, E and F

The analysis for data set A does not hold for the other data sets. The Figures 70 and 71 shows the results for data set C, which is representative for the rest of the data sets. The components in these sets consists mainly of either one

node or many unary nodes. When changing the percentage, the new giraffe trees are more or less the same as before.

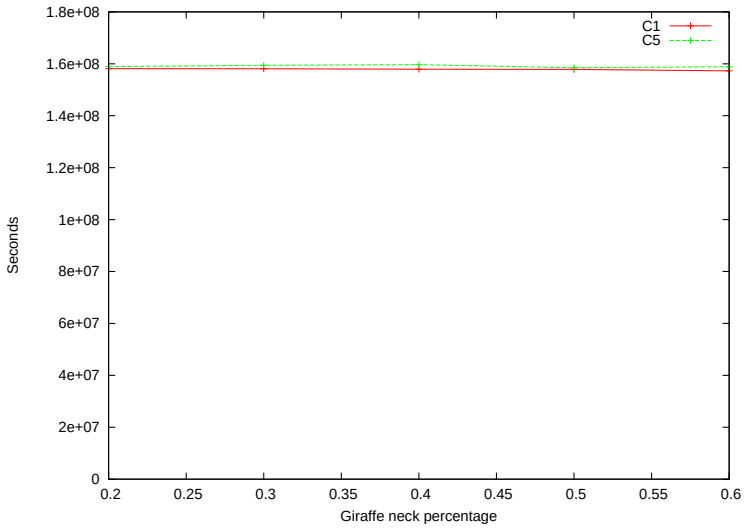
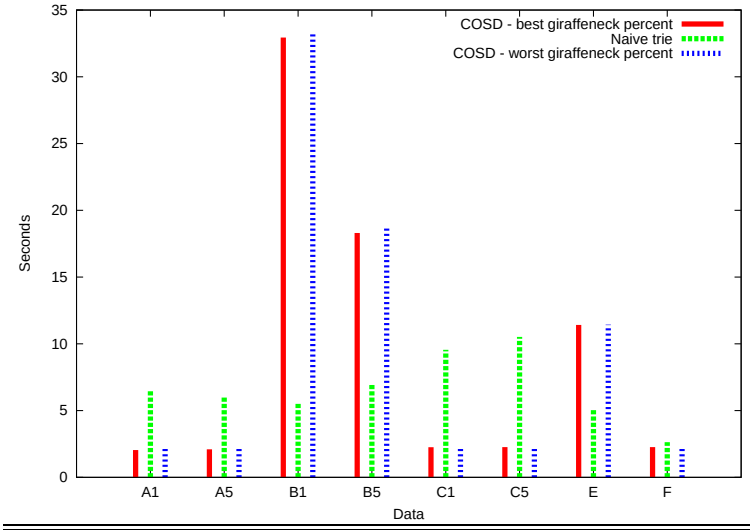


Figure 71: The graph shows the level 1 cache misses for the cache oblivious string dictionary for various giraffe trees for data set C.

22.3 Comparison of execution times

The comparison of executions time for the giraffe experiments yeilds the same results as the comparison for various ε. Again long path of unary nodes favors the cache oblivious string dictionary search algorithm. In the data sets with many bridge nodes the overhead of searching in these nodes is too large for the cache oblivious string dictionary. Figure 72 shows the execution times for experiments on the **Internal** computer.

When performing the experiments on the **Swap** computer, the results are the same with the ε experiments. Choosing a suitable percentage of nodes as ancestors makes searching in the cache oblivious string dictionary faster than in the naive trie. Figure 73 illustrates this.



Name	COSD, best	Naive trie	COSD, worst
D1	42.0	799.9	42.1
D5	30.7	319.7	32.5

Figure 72: Execution time for searching in the cache oblivious string dictionary (best and worst percentage of giraffeneck) and the naive trie.

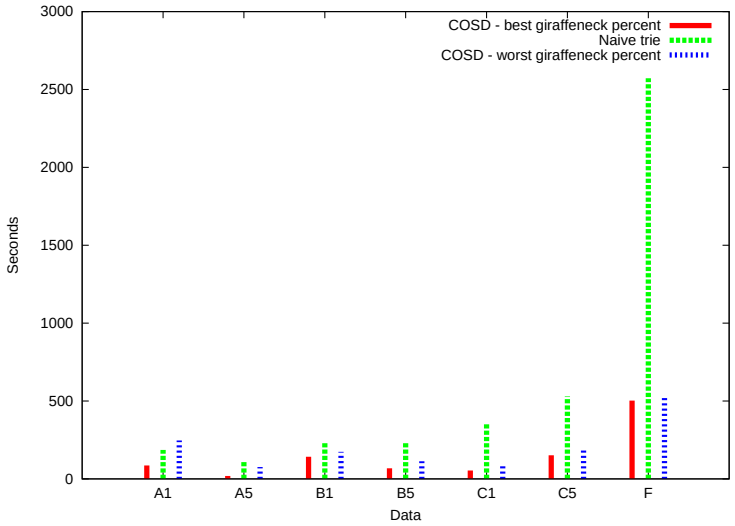


Figure 73: Execution time for searching in the cache oblivious string dictionary (best and worst percentage of giraffeneck) and the naive trie.on the **Swap** computer.

22.4 Conclusion

The change of giraffe neck percentage does not effect the execution time much when searching in the cache oblivious string dictionary. This is probably because the components are fairly small and therefore the giraffe trees do not change much when varying the ancestor percentage. In some experiments there are small variations in the number of cache misses. This do not seem to influence the execution time which is most likely due to prefetching.

23 Construction time

The previous sections shows that searching in the cache oblivious string dictionary can be faster than searching in the naive trie. Especially when data is put in swap memory.

However, in the previous sections the construction time is not included in the results. As both the naive trie and cache oblivious string dictionary needs to build their structure before searching, this is an important issue. Figure 74 shows the construction time for the naive trie structure and the cache oblivious string dictionary layout. The cache oblivious string dictionary layout is created with $\varepsilon = 0.5$, giraffe percentage at 0.5 (50%), Huffman trees and the article tree connecting the components.

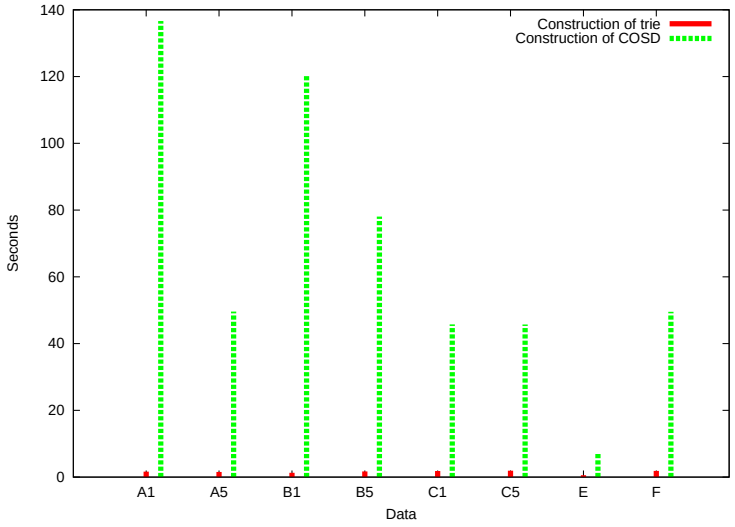


Figure 74: Comparison of construction time between the naive trie and the cache oblivious string dictionary. The cache oblivious string dictionary layout is created with $\varepsilon = 0.5$, giraffe percentage at 0.5 (50%), Huffman trees and the article tree connecting the components.

The experiments so far conclude that the `corunner` is superier to the `trierunner` for almost all of the data sets, and all when needing to swap. The `corunner` is not superier however when factoring in the time to construct the layout of the cache oblivious string dictonary.

As seen on Figure 74 the construction time of the cache oblivious layout is much more time consuming. This can be outweighed if there will be many searches for a constructed dictonary.

The figure shows that the construction time for the cache oblivious string dictionary are much worse than for the naive trie. This means that if search-

Name	Construction of trie	Construction of COSD
D1	77.9	1131.4
D5	31.5	519.7

Table 9: Comparison of construction time between the naive trie and the cache oblivious string dictionary. The cache oblivious string dictionary layout is created with $\varepsilon = 0.5$, giraffe percentage at 0.5 (50%), Huffman trees and the article tree connecting the components.

ing in the data is only done a few times, the cache oblivious string dictionary cannot compete with the naive trie. Even if the naive trie has to construct is structure each time.

24 Strings with errors

There are not much difference in the behaviour between the experiments using the strings with error and those without. The execution time is of cause smaller as no string is searched fully.

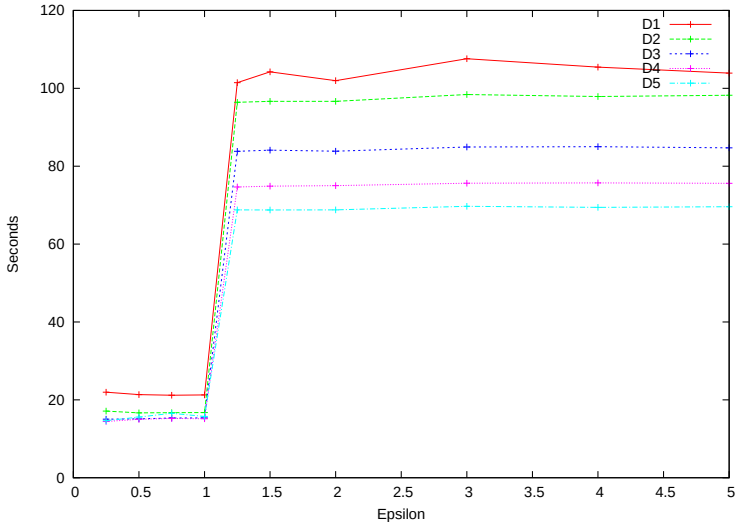


Figure 75: Execution times for the cache oblivious string dictionary on error string from data set D for various ε.

Figure 75 show the execution time for the data set D using the strings with errors. Compared with figure 59 where the strings without errors are used, the same patterns can be found.

The graph for data set D is chosen as representative for the rest of the data sets. All the graphs where the strings with errors are used show the same patterns as the graphs where the string without errors are used. The only difference is the execution time, which is smaller when the strings with errors are used.

The comparison between the cache oblivious string dictionary and the naive trie using strings with errors is shown in figure 76. Again there are no significant difference between searching for strings with errors and searching for strings without errors, Figure 65. The only difference is smaller execution time. Even though only graphs for variation of ε is shown, the same applies for the variation of the giraffe neck percentage.

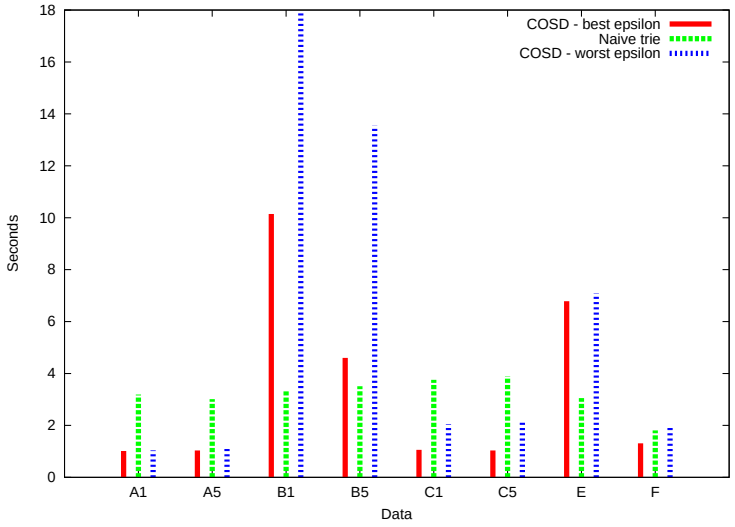


Figure 76: Execution times using the cache oblivious string dictionary (best and worst ε) and the naive trie both searching for strings with errors.

25 Final conclusion

In this thesis the cache oblivious string dictionary have been implemented. It is compared to a naive trie structure to see if the cache oblivious approach would yield any significant improvements in execution time.

The experiments show that the cache oblivious string dictionary can perform better than the naive trie. Depending on the type of data different parameters need to be right for the cache oblivious string dictionary to perform optimal.

When changing the ε value the change in execution time is notable. It is not possible to choose a fixed ε for all data types. For each data type, an ε value needs to found, as the value sometime must be small and some times large. The experiments show that small values of ε is mostly preferable. The time gain from moving to larger ε for the data sets requiring it, is smaller then the loss for the sets not requiring it.

When changing the number of ancestor nodes in a giraffe tree the experiments show no real difference in execution time. The only notable difference is the space usage which increases as the number of ancestor nodes grows.

As the construction algorithm in this thesis is not cache oblivious, different weight balanced trees have been examined as replacements for the article tree. Both the Huffman tree and the leaf oriented optimal binary search tree performs better than the article tree, but only at a fraction.

When moving to swap the experiments perform significantly better than the naive trie structure. This is the result of increased access time, which influences the the cache oblivious layout less than the trie.

Experimenting with string with errors both in memory and swap yielded no interesting results, as the performance difference is similar.

Before choosing the cache oblivious string dictionary structure over the naive trie structure, the construction time of the cache oblivious layout needs to be addressed. The experiments show that the construction time of the cache oblivious string dictionary is significantly larger than the naive trie. Furthermore, it is not possible to delete or insert new strings into the cache oblivious string dictionary.

It can be concluded that the cache oblivious string dictionary structure can outperform the naive trie both in memory and swap. However, this is only possible when a large amount of searches are performed.

25.1 Future work

In the future is would be interesting to tune the trie structure. This could be done by a van Emde Boas layout of the trie or a simple BFS layout. It

would be interesting to examine whether or not the cache oblivious string dictionary still perform well in comparisons.

The construction time is also an issue. Because of the large overhead, the structure is not competitive in a wide variety of applications. Reducing the overhead, by for instance storing the component in different files, and then process them one by one until the final layout is possible, could make the structure a whole lot more interesting.

Part V

Appendix

A Source code

The source code can be downloaded at the web address

`http://www.daimi.au.dk/~stumme/Download/Thesis-sourcecode.tar.bz`

To extract the code from the file type

```
tar -xvzf Thesis-sourcecode.tar.bz
```

A folder named `Thesis` will be created. To compile the source code type (inside the `Thesis` folder) `make`. Three programs will be created. One for running the naive trie, `trierunner`, one for doing the cache oblivious layout, `colayouter`, and one for searching in the cache oblivious layout, `corunner`.

A.1 Input file format

The `trierunner` and `colayouter` program both requires an input file. This file consists of ASCII strings, one on each line. The only character values not allowed is 0-33, 69, 127 and 128 as these are special purpose characters.

A.2 The naive trie program

The trie program takes several parameters as input. These are shown in Figure 77.

```
trierunner -i <INPUT_FILE> <ARGUMENTS*>
       -i <INPUT-FILE> (REQUIRED)
       -t <TEST-FILE> (REQUIRED)
       -c <CHILD-TREE> (Standard 0) 0 : Red-Black Tree, 1 : Sorted Vector
       -v <SHOW INFO> (Standard 0) 0 : None, 1 : All
       -r <REPETITION OF TEST-FILE> (Standard 1) Values must be an integer above zero
```

Figure 77: The arguments possible for the `trierunner` program.

INPUT-FILE An input file as described in section A.1.

TEST-FILE A file containing strings as described in section A.1. These strings are then search for in the trie.

CHILD-TREE The search tree used to store the children.

SHOW INFO Displays the process of the program.

REPETITION OF TEST-FILE If a test file is given this arguments indicates how many times the strings in the test file is search for. It is done in a round robin fashion.

The output of the program is

1. Number of trie nodes.
2. Total size of trie structure (in bytes).
3. Construction time (in seconds).
4. Running time (in seconds).
5. Number of search strings found.
6. Number of search strings not found.

in that order.

A.3 The cache oblivious layout program

The layout program takes several parameters as input. Only the input file is required. The parameters are shown in Figure 78.

```
colayout -i <INPUT-FILE> <ARGUMENTS*>
       -i <INPUT-FILE> (REQUIRED)
       -o <OUTPUT-FILE> (If none, output is:
           layout-<INPUT-FILE>-<EPSILON>-<LAYER-MULTIPLIER>-
                  <COMPONENT-TREE>-<BRIDGE-ALGORITHM>.veb)
       -e <EPSILON> (Standard: 0.5)
       -b <BRIDGE-ALGORITHM> (Standard: 0)
                           0 : Weight balanced search tree,
                           1 : Optimal search tree
       -c <COMPONENT-TREE> (Standard: 0)
                           0 : Huffman,
                           1 : Weight balanced search tree
       -v <SHOW INFO> (Standard: 0) 0 : None, 1 : All
       -w <WRITE-METAPOST-LATEX> (Standard: 0) 0 : No, 1 : Yes
       -d <% OF GIRAFFE> (Standard: 0.5) Used in giraffe tree
```

Figure 78: The arguments possible for the `colayouter` program.

INPUT-FILE An input file as described in section A.1.

OUTPUT-FILE The layout of the input file. A predefined is used if no argument is given[41].

EPSILON The value of ε.

BRIDGE-ALGORITHM The tree algorithm used to connect the components.

COMPONENT-TREE The tree algorithm used inside the components.

SHOW INFO Displays the process of the program.

WRITE-METAPOST-LATEX Creates a Metapost file and a associated latex file for all trees in the layout. Note, that the Metapost algorithm for drawing the trees takes long time for even small trees.

% OF GIRAFFE Indicates how many procentage of the nodes are ancestors before it is a legal giraffe tree. The given parameter $\in (0.0; 1.0)$.

The output of the program to standard out is

1. Total space usage (in bytes).
2. Blind trie space usage (in bytes).
3. Giraffe tree space usage (in bytes).
4. Bridge space usage (in bytes).
5. Construction time (in seconds).
6. Number of trie nodes.
7. Number of components.
8. Average number of nodes inside components.
9. Number of blind tries.
10. Number of giraffe trees.
11. Number of bridges.

in that order.

[41] See Figure 78 for the predefined file name.

A.4 The cache oblivious search program

The cache oblivious search program takes several parameters as input. Only the input file is required. The parameters are shown in Figure 79.

```
corunner -i <INPUT-FILE> <ARGUMENTS*>
       -i <INPUT-FILE> (REQUIRED veb-file)
       -t <TEST-FILE> (If none given, user interaction is possible)
       -v <SHOW INFO> (Standard 0) 0 : None, 1 : All
       -r <REPETITION OF TEST-FILE> (Standard 1)
                            Values must be an integer above zero
       -c <COUNTER ON> (Standard: 0) 0 : No, 1 : Yes
```

Figure 79: The arguments possible for the `corunner` program.

INPUT-FILE An input file created by the `colayouter` program.

TEST-FILE A file containing strings as described in section A.1. These strings are then search for in the trie. If no file is given, it is possible to type search strings.

SHOW INFO Displays the process of the program.

REPETITION OF TEST-FILE The number of times the strings in the test file is tested. This is done in a round robin fashion.

COUNTER ON Displays number of searches completed of the total number of searches.

The output to standard out is

1. Load time (in seconds).
2. Running time (in seconds).
3. The number of strings found.
4. The number of strings not found.

in that order.

B Construction tables

In this section the output from the layout program is displayed. The different shortening is briefly explained below.

ε Value of ε.

% Procentage of nodes needed as ancestors for a valig giraffe tree.

No. TN Number of trie nodes.

No. C Number of components.

No. BT Number of blind tries.

No. G Number of giraffe trees.

No. B Number of bridges.

Nodes / C Number of nodes per component (Average).

BT space Total space usage for the blind tries nodes (in bytes).

G space Total space usage for the giraffe tree nodes (in bytes).

B space Total space usage for the bridge nodes (in bytes).

Total space Total space usage (in bytes).

Cons. time Construction time for the layout (in seconds).

Data for A1 for varying ε											
ε	**No. TN**	**No. C**	**No. BT**	**No. G**	**No. B**	**Nodes / C**	**BT space**	**G space**	**B space**	**Total space**	**Cons. time**
0.25	3014010	5427	26229	27756	1973	555.373	573580	73829500	142064	74545144	127.552
0.500	3014010	186	930	4099	71	16204.400	172460	105646220	4784	105823464	136.645
0.750	3014010	5195	25179	25566	1878	580.175	557400	81125180	136160	81818740	129.547
1.000	3014010	1	5	1115	0	3014010.000	157820	127661060	0	127818880	145.666
1.250	3014010	1	5	1115	0	3014010.000	157820	127661060	0	127818880	144.971
1.500	3014010	1	5	1115	0	3014010.000	157820	127661060	0	127818880	145.185
2.000	3014010	1	5	1115	0	3014010.000	157820	127661060	0	127818880	145.050
3.000	3014010	1	5	1115	0	3014010.000	157820	127661060	0	127818880	144.956
4.000	3014010	1	5	1115	0	3014010.000	157820	127661060	0	127818880	144.964
5.000	3014010	1	5	1115	0	3014010.000	157820	127661060	0	127818880	144.998

Data for A2 for varying ε											
ε	**No. TN**	**No. C**	**No. BT**	**No. G**	**No. B**	**Nodes / C**	**BT space**	**G space**	**B space**	**Total space**	**Cons. time**
0.250	3379858	8149	39330	41632	2969	414.757	858700	79995100	213232	81067032	111.497
0.500	3379858	279	1395	6148	107	12114.200	257400	112361600	7184	112626184	121.580
0.750	3379858	7821	37808	38380	2837	432.152	835100	90514100	204848	91554048	115.080
1.000	3379858	1	5	1671	0	3379860.000	235440	141512840	0	141748280	131.468
1.250	3379858	1	5	1671	0	3379860.000	235440	141512840	0	141748280	131.521
1.500	3379858	1	5	1671	0	3379860.000	235440	141512840	0	141748280	131.374
2.000	3379858	1	5	1671	0	3379860.000	235440	141512840	0	141748280	131.372
3.000	3379858	1	5	1671	0	3379860.000	235440	141512840	0	141748280	131.677
4.000	3379858	1	5	1671	0	3379860.000	235440	141512840	0	141748280	131.213
5.000	3379858	1	5	1671	0	3379860.000	235440	141512840	0	141748280	131.274

Data for A3 for varying ε											
ε	**No. TN**	**No. C**	**No. BT**	**No. G**	**No. B**	**Nodes / C**	**BT space**	**G space**	**B space**	**Total space**	**Cons. time**
0.250	3377396	13523	54108	54916	5099	249.752	1115920	70799240	351120	72266280	77.910
0.500	3377396	387	2610	11517	140	8727.120	358780	132504960	10112	132873852	96.521
0.750	3377396	3335	23342	23711	1261	1012.710	711480	76462120	86512	77260112	78.983
1.000	3377396	1	6	2227	0	3377400.000	314440	143695440	0	144009880	101.460
1.250	3377396	1	6	2227	0	3377400.000	314440	143695440	0	144009880	101.386
1.500	3377396	1	6	2227	0	3377400.000	314440	143695440	0	144009880	101.057
2.000	3377396	1	6	2227	0	3377400.000	314440	143695440	0	144009880	101.451
3.000	3377396	1	6	2227	0	3377400.000	314440	143695440	0	144009880	101.236
4.000	3377396	1	6	2227	0	3377400.000	314440	143695440	0	144009880	101.043
5.000	3377396	1	6	2227	0	3377400.000	314440	143695440	0	144009880	101.493

Data for A4 for varying ε											
ε	**No. TN**	**No. C**	**No. BT**	**No. G**	**No. B**	**Nodes / C**	**BT space**	**G space**	**B space**	**Total space**	**Cons. time**
0.250	3281877	17847	71495	72113	6789	183.890	1449680	67047200	462448	68959328	56.323
0.500	3281877	480	3261	15282	171	6837.240	448520	118989880	12592	119450992	71.471
0.750	3281877	4168	29174	37378	1569	787.399	889440	98395820	108240	99393500	65.159
1.000	3281877	1	6	2782	0	3281880.000	393020	136527100	0	136920120	78.911
1.250	3281877	1	6	2782	0	3281880.000	393020	136527100	0	136920120	79.223
1.500	3281877	1	6	2782	0	3281880.000	393020	136527100	0	136920120	78.285
2.000	3281877	1	6	2782	0	3281880.000	393020	136527100	0	136920120	78.628
3.000	3281877	1	6	2782	0	3281880.000	393020	136527100	0	136920120	78.280
4.000	3281877	1	6	2782	0	3281880.000	393020	136527100	0	136920120	78.480
5.000	3281877	1	6	2782	0	3281880.000	393020	136527100	0	136920120	78.545

Data for A5 for varying ε											
ε	**No. TN**	**No. C**	**No. BT**	**No. G**	**No. B**	**Nodes / C**	**BT space**	**G space**	**B space**	**Total space**	**Cons. time**
0.250	2812226	20918	82843	84082	7860	134.440	1696020	58945760	543584	61185364	43.797
0.500	2812226	586	7258	18430	215	4799.020	604980	82865060	15280	83485320	49.585
0.750	2812226	5001	20008	30561	1917	562.333	766680	84234120	129328	85130128	50.146
1.000	2812226	1	8	4987	0	2812230.000	471680	144983420	0	145455100	69.715
1.250	2812226	1	8	4987	0	2812230.000	471680	144983420	0	145455100	69.997
1.500	2812226	1	8	4987	0	2812230.000	471680	144983420	0	145455100	70.131
2.000	2812226	1	8	4987	0	2812230.000	471680	144983420	0	145455100	69.999
3.000	2812226	1	8	4987	0	2812230.000	471680	144983420	0	145455100	69.833
4.000	2812226	1	8	4987	0	2812230.000	471680	144983420	0	145455100	69.840
5.000	2812226	1	8	4987	0	2812230.000	471680	144983420	0	145455100	69.963

Data for B1 for varying ε											
ε	**No. TN**	**No. C**	**No. BT**	**No. G**	**No. B**	**Nodes / C**	**BT space**	**G space**	**B space**	**Total space**	**Cons. time**
0.250	2019486	1849078	1849078	1849078	618172	1.092	36981560	40389740	49279712	126651012	121.357
0.500	2019486	1849078	1849078	1849078	618172	1.092	36981560	40389740	49279712	126651012	120.534
0.750	2019486	1849067	1849077	1849078	618171	1.092	36981580	40389780	49279376	126650736	122.061
1.000	2019486	1849067	1849077	1849078	618171	1.092	36981580	40389780	49279376	126650736	121.354
1.250	2019486	1647177	1655029	1770161	562516	1.226	36992740	42739100	43709376	123441216	114.659
1.500	2019486	1647171	1655024	1770161	562515	1.226	36992760	42739220	43709200	123441180	114.496
2.000	2019486	1605981	1657938	1807970	544913	1.257	37891900	44425140	42672752	124989792	112.894
3.000	2019486	16242	332118	1094488	5693	124.337	43359980	67428200	428624	111216804	55.061
4.000	2019486	48783	358448	1139128	15873	41.397	43225320	67055140	1307056	111587516	55.476
5.000	2019486	1216901	1217007	1325887	409877	1.660	37030960	54216880	32382768	123630608	138.352

Data for B2 for varying ε											
ε	**No. TN**	**No. C**	**No. BT**	**No. G**	**No. B**	**Nodes / C**	**BT space**	**G space**	**B space**	**Total space**	**Cons. time**
0.250	3078176	1139111	1562486	1562486	390508	2.702	31249720	61563540	30203392	123016652	101.815
0.500	3078176	1139111	1562486	1562486	390508	2.702	31249720	61563540	30203392	123016652	101.448
0.750	3078176	1034633	1501049	1559959	359136	2.975	31854560	63346700	27362048	122563308	98.021
1.000	3078176	1077949	1529046	1559989	373970	2.856	31545280	62501240	28510816	122557336	99.194
1.250	3078176	532365	1139591	1527265	183749	5.782	34836600	74010200	14095664	122942464	83.234
1.500	3078176	350025	477810	992597	118587	8.794	25356060	86149060	9303376	120808496	71.901
2.000	3078176	237326	913658	1469043	82295	12.970	36268860	80504040	6277680	123050580	72.425
3.000	3078176	3050	15726	538498	1055	1009.240	23061200	120673920	80688	143815808	61.990
4.000	3078176	1	6490	523814	0	3078180.000	22938380	124877900	0	147816280	64.338
5.000	3078176	1	6490	523814	0	3078180.000	22938380	124877900	0	147816280	64.513

Data for B3 for varying ε											
ε	**No. TN**	**No. C**	**No. BT**	**No. G**	**No. B**	**Nodes / C**	**BT space**	**G space**	**B space**	**Total space**	**Cons. time**
0.250	2286364	759600	1504644	1504644	260495	3.010	30092880	45727300	20139248	95959428	77.053
0.500	2286364	757859	1503757	1504630	259629	3.017	30109920	45761820	20097392	95969132	76.968
0.750	2286364	674787	1427277	1475145	232305	3.388	30067460	46845880	17876272	94789612	73.109
1.000	2286364	700989	1431010	1460914	240708	3.262	29614520	46830940	18580288	95025748	73.825
1.250	2286364	180144	847927	1227296	61033	12.692	28560400	56566600	4788048	89915048	54.286
1.500	2286364	497013	1032868	1199775	168031	4.600	25925260	58599120	13215888	97740268	65.715
2.000	2286364	163892	822140	1176875	55458	13.950	28373220	55867480	4357184	88597884	53.708
3.000	2286364	1	250	171985	0	2286360.000	15214800	95698640	0	110913440	47.100
4.000	2286364	1	250	171985	0	2286360.000	15214800	95698640	0	110913440	47.301
5.000	2286364	1	250	171985	0	2286360.000	15214800	95698640	0	110913440	47.093

Data for B4 for varying ε											
ε	**No. TN**	**No. C**	**No. BT**	**No. G**	**No. B**	**Nodes / C**	**BT space**	**G space**	**B space**	**Total space**	**Cons. time**
0.250	2753553	645854	1337054	1337054	221573	4.263	26741080	55071080	17122128	98934288	76.454
0.500	2753553	588747	1251119	1285208	203821	4.677	26059220	56108100	15578736	97746056	73.288
0.750	2753553	221261	632110	910185	76542	12.445	21104400	63526160	5855648	90486208	56.332
1.000	2753553	466654	1036621	1150292	162110	5.901	24199520	59261420	12339136	95800076	67.731
1.250	2753553	28659	125886	513904	9919	96.080	14869020	85558880	758352	101186252	49.824
1.500	2753553	339806	746166	845133	114923	8.103	21057840	67503620	9034992	97596452	62.097
2.000	2753553	186073	546328	806186	62936	14.798	20136540	68785540	4947328	93869408	56.502
3.000	2753553	1	88	139365	0	2753550.000	12933200	124218220	0	137151420	57.120
4.000	2753553	1	88	139365	0	2753550.000	12933200	124218220	0	137151420	57.085
5.000	2753553	1	88	139365	0	2753550.000	12933200	124218220	0	137151420	56.895

Data for B5 for varying ε											
ε	**No. TN**	**No. C**	**No. BT**	**No. G**	**No. B**	**Nodes / C**	**BT space**	**G space**	**B space**	**Total space**	**Cons. time**
0.250	3195515	569757	1681091	1681091	195390	5.609	33621820	63910320	15105952	112638092	82.459
0.500	3195515	517990	1538207	1569184	179269	6.169	31706140	64550940	13707344	109964424	78.064
0.750	3195515	185899	555960	807594	64285	17.189	18774060	69121560	4920176	92815796	56.102
1.000	3195515	417098	1243333	1340105	144702	7.661	27816100	67067440	11031872	105915412	72.407
1.250	3195515	62540	222036	525815	21434	51.096	14580900	108508220	1658304	124747424	59.924
1.500	3195515	30636	248679	532107	10455	104.306	15765660	88378900	813040	104957600	53.576
2.000	3195515	162867	491625	722001	55628	19.620	17966200	76511120	4321664	98798984	57.154
3.000	3195515	2	53	92058	1	1597760.000	11408820	123860000	16	135268836	58.178
4.000	3195515	1	34	92058	0	3195520.000	11408460	124646960	0	136055420	58.100
5.000	3195515	1	34	92058	0	3195520.000	11408460	124646960	0	136055420	57.543

Data for C1 for varying ε											
ε	**No. TN**	**No. C**	**No. BT**	**No. G**	**No. B**	**Nodes / C**	**BT space**	**G space**	**B space**	**Total space**	**Cons. time**
0.250	2811774	18467	67301	67301	4591	152.259	1346020	56235500	517456	58098976	47.468
0.500	2811774	18458	67276	67276	4591	152.334	1345520	56235500	517168	58098188	45.742
0.750	2811774	17401	66077	66227	4441	161.587	1353700	56241420	485744	58080864	46.094
1.000	2811774	4	16	3350	1	702944.000	391800	121690240	80	122082120	69.083
1.250	2811774	1	13	3348	0	2811770.000	391800	121732540	0	122124340	68.848
1.500	2811774	1	13	3348	0	2811770.000	391800	121732540	0	122124340	68.636
2.000	2811774	1	13	3348	0	2811770.000	391800	121732540	0	122124340	68.858
3.000	2811774	1	13	3348	0	2811770.000	391800	121732540	0	122124340	68.698
4.000	2811774	1	13	3348	0	2811770.000	391800	121732540	0	122124340	68.484
5.000	2811774	1	13	3348	0	2811770.000	391800	121732540	0	122124340	69.083

Data for C2 for varying ε											
ε	**No. TN**	**No. C**	**No. BT**	**No. G**	**No. B**	**Nodes / C**	**BT space**	**G space**	**B space**	**Total space**	**Cons. time**
0.250	2825918	19277	68051	68051	5030	146.595	1361020	56518380	536352	58415752	47.440
0.500	2825918	19269	68033	68033	5030	146.656	1360660	56518380	536096	58415136	46.290
0.750	2825918	17716	66345	66482	4895	159.512	1370420	56564800	488560	58423780	47.590
1.000	2825918	3	11	3380	1	941973.000	400540	122550980	48	122951568	69.472
1.250	2825918	1	9	3378	0	2825920.000	400540	122598320	0	122998860	69.508
1.500	2825918	1	9	3378	0	2825920.000	400540	122598320	0	122998860	69.612
2.000	2825918	1	9	3378	0	2825920.000	400540	122598320	0	122998860	69.519
3.000	2825918	1	9	3378	0	2825920.000	400540	122598320	0	122998860	69.742
4.000	2825918	1	9	3378	0	2825920.000	400540	122598320	0	122998860	69.694
5.000	2825918	1	9	3378	0	2825920.000	400540	122598320	0	122998860	69.971

Data for C3 for varying ε											
ε	**No. TN**	**No. C**	**No. BT**	**No. G**	**No. B**	**Nodes / C**	**BT space**	**G space**	**B space**	**Total space**	**Cons. time**
0.250	2844036	19325	68187	68187	5080	147.169	1363740	56880740	537088	58781568	47.011
0.500	2844036	19319	68171	68171	5080	147.214	1363420	56880740	536896	58781056	46.193
0.750	2844036	18713	67434	67564	4950	151.982	1366120	56885840	519584	58771544	46.634
1.000	2844036	2	10	3401	1	1422020.000	401480	123326400	16	123727896	69.886
1.250	2844036	1	9	3400	0	2844040.000	401500	123328580	0	123730080	69.769
1.500	2844036	1	9	3400	0	2844040.000	401500	123328580	0	123730080	69.478
2.000	2844036	1	9	3400	0	2844040.000	401500	123328580	0	123730080	70.094
3.000	2844036	1	9	3400	0	2844040.000	401500	123328580	0	123730080	69.737
4.000	2844036	1	9	3400	0	2844040.000	401500	123328580	0	123730080	69.731
5.000	2844036	1	9	3400	0	2844040.000	401500	123328580	0	123730080	70.989

Data for C4 for varying ε											
ε	**No. TN**	**No. C**	**No. BT**	**No. G**	**No. B**	**Nodes / C**	**BT space**	**G space**	**B space**	**Total space**	**Cons. time**
0.250	2817974	19498	68307	68307	5260	144.526	1366140	56359500	539744	58265384	46.805
0.500	2817974	19492	68291	68291	5260	144.571	1365820	56359500	539552	58264872	46.405
0.750	2817974	18679	67380	67477	5161	150.863	1371760	56363300	515120	58250180	46.416
1.000	2817974	2	10	3358	1	1408990.000	405000	122098920	16	122503936	70.150
1.250	2817974	2	10	3358	1	1408990.000	405000	122098920	16	122503936	70.038
1.500	2817974	2	10	3358	1	1408990.000	405000	122098920	16	122503936	69.547
2.000	2817974	2	10	3358	1	1408990.000	405000	122098920	16	122503936	69.533
3.000	2817974	1	9	3358	0	2817970.000	405020	122106480	0	122511500	69.777
4.000	2817974	1	9	3358	0	2817970.000	405020	122106480	0	122511500	69.992
5.000	2817974	1	9	3358	0	2817970.000	405020	122106480	0	122511500	69.910

Data for C5 for varying ε											
ε	**No. TN**	**No. C**	**No. BT**	**No. G**	**No. B**	**Nodes / C**	**BT space**	**G space**	**B space**	**Total space**	**Cons. time**
0.250	2816441	20006	68733	68733	5412	140.780	1374660	56328840	553568	58257068	46.525
0.500	2816441	20002	68724	68724	5412	140.808	1374480	56328840	553440	58256760	45.735
0.750	2816441	19311	67907	68037	5282	145.846	1377120	56333880	533408	58244408	46.412
1.000	2816441	1	5	3359	0	2816440.000	407720	122154940	0	122562660	70.178
1.250	2816441	1	5	3359	0	2816440.000	407720	122154940	0	122562660	70.468
1.500	2816441	1	5	3359	0	2816440.000	407720	122154940	0	122562660	70.334
2.000	2816441	1	5	3359	0	2816440.000	407720	122154940	0	122562660	70.782
3.000	2816441	1	5	3359	0	2816440.000	407720	122154940	0	122562660	70.429
4.000	2816441	1	5	3359	0	2816440.000	407720	122154940	0	122562660	70.277
5.000	2816441	1	5	3359	0	2816440.000	407720	122154940	0	122562660	70.886

Data for D1 for varying ε											
ε	**No. TN**	**No. C**	**No. BT**	**No. G**	**No. B**	**Nodes / C**	**BT space**	**G space**	**B space**	**Total space**	**Cons. time**
0.250	254051	123672	123924	123924	123550	2.054	2478480	5081040	1980672	9540192	1137.300
0.500	254051	123602	123717	123726	123541	2.055	2474700	5102640	1978576	9555916	1131.390
0.750	254051	123220	123294	123445	123199	2.062	2479920	5132240	1971824	9583984	1137.260
1.000	254051	123570	123586	123603	123530	2.056	2472520	5115080	1977728	9565328	1137.500
1.250	254051	1100	1114	1565	1099	230.955	4920320	14480940	17584	19418844	1242.340
1.500	254051	1100	1114	1565	1099	230.955	4920320	14480940	17584	19418844	1248.190
2.000	254051	1100	1114	1565	1099	230.955	4920320	14480940	17584	19418844	1247.100
3.000	254051	1	15	494	0	254051.000	4942300	14545400	0	19487700	1248.590
4.000	254051	1	15	494	0	254051.000	4942300	14545400	0	19487700	1241.760
5.000	254051	1	15	494	0	254051.000	4942300	14545400	0	19487700	1244.630

Data for D2 for varying ε											
ε	**No. TN**	**No. C**	**No. BT**	**No. G**	**No. B**	**Nodes / C**	**BT space**	**G space**	**B space**	**Total space**	**Cons. time**
0.250	254550	123743	124136	124136	123549	2.057	2482720	5091020	1982960	9556700	700.148
0.500	254550	123619	123767	123789	123527	2.059	2476220	5250000	1979344	9705564	702.066
0.750	254550	123048	123149	123377	123017	2.069	2484260	5283340	1969232	9736832	703.800
1.000	254550	123581	123596	123623	123520	2.060	2473080	5258460	1978240	9709780	701.465
1.250	254550	819	830	1273	818	310.806	4925840	14444400	13088	19383328	758.293
1.500	254550	819	830	1273	818	310.806	4925840	14444400	13088	19383328	754.081
2.000	254550	819	830	1273	818	310.806	4925840	14444400	13088	19383328	755.277
3.000	254550	1	12	474	0	254550.000	4942200	14491960	0	19434160	755.186
4.000	254550	1	12	474	0	254550.000	4942200	14491960	0	19434160	761.600
5.000	254550	1	12	474	0	254550.000	4942200	14491960	0	19434160	754.948

Data for D3 for varying ε											
ε	**No. TN**	**No. C**	**No. BT**	**No. G**	**No. B**	**Nodes / C**	**BT space**	**G space**	**B space**	**Total space**	**Cons. time**
0.250	255018	123797	124327	124328	123516	2.060	2486580	5100800	1985216	9572596	556.414
0.500	255018	123612	123790	123823	123485	2.063	2477120	5501180	1979792	9958092	555.096
0.750	255018	122854	122984	123286	122813	2.076	2487880	5540760	1966288	9994928	553.194
1.000	255018	123558	123571	123609	123478	2.064	2473020	5507500	1978176	9958696	553.509
1.250	255018	692	702	1139	691	368.523	4927100	14440200	11056	19378356	589.385
1.500	255018	692	702	1139	691	368.523	4927100	14440200	11056	19378356	593.388
2.000	255018	692	702	1139	691	368.523	4927100	14440200	11056	19378356	596.108
3.000	255018	1	11	465	0	255018.000	4940920	14486060	0	19426980	596.068
4.000	255018	1	11	465	0	255018.000	4940920	14486060	0	19426980	598.134
5.000	255018	1	11	465	0	255018.000	4940920	14486060	0	19426980	594.071

Data for D4 for varying ε											
ε	**No. TN**	**No. C**	**No. BT**	**No. G**	**No. B**	**Nodes / C**	**BT space**	**G space**	**B space**	**Total space**	**Cons. time**
0.250	255450	123797	124453	124455	123448	2.063	2489140	5110780	1986304	9586224	508.702
0.500	255450	123564	123781	123824	123407	2.067	2477340	5852940	1979504	10309784	506.993
0.750	255450	122617	122777	123157	122566	2.083	2490900	5902460	1962656	10356016	511.283
1.000	255450	123500	123512	123560	123400	2.068	2472240	5859760	1977568	10309568	509.114
1.250	255450	654	664	1089	653	390.596	4925160	14244880	10448	19180488	539.617
1.500	255450	654	664	1089	653	390.596	4925160	14244880	10448	19180488	539.181
2.000	255450	654	664	1089	653	390.596	4925160	14244880	10448	19180488	539.280
3.000	255450	1	11	452	0	255450.000	4938220	14285740	0	19223960	539.148
4.000	255450	1	11	452	0	255450.000	4938220	14285740	0	19223960	538.839
5.000	255450	1	11	452	0	255450.000	4938220	14285740	0	19223960	542.896

Data for D5 for varying ε											
ε	**No. TN**	**No. C**	**No. BT**	**No. G**	**No. B**	**Nodes / C**	**BT space**	**G space**	**B space**	**Total space**	**Cons. time**
0.250	256019	123931	124700	124702	123517	2.066	2494080	5122260	1989488	9605828	520.259
0.500	256019	123652	123910	123963	123466	2.070	2480320	6307260	1981376	10768956	519.726
0.750	256019	113560	113750	114289	113500	2.254	2675760	7033680	1817888	11527328	519.504
1.000	256019	123573	123589	123650	123456	2.072	2474300	6314520	1979008	10767828	518.955
1.250	256019	623	633	1047	622	410.945	4928520	14103920	9952	19042392	544.774
1.500	256019	623	633	1047	622	410.945	4928520	14103920	9952	19042392	550.163
2.000	256019	623	633	1047	622	410.945	4928520	14103920	9952	19042392	550.570
3.000	256019	1	11	440	0	256019.000	4940960	14142080	0	19083040	550.063
4.000	256019	1	11	440	0	256019.000	4940960	14142080	0	19083040	548.438
5.000	256019	1	11	440	0	256019.000	4940960	14142080	0	19083040	545.304

Data for E for varying ε											
ε	**No. TN**	**No. C**	**No. BT**	**No. G**	**No. B**	**Nodes / C**	**BT space**	**G space**	**B space**	**Total space**	**Cons. time**
0.250	143617	69998	94803	94803	20130	2.052	1896060	2872360	1917824	6686244	7.327
0.500	143617	67815	92998	93791	19160	2.118	1898840	2899520	1863488	6661848	7.172
0.750	143617	59972	87207	91439	16187	2.395	1919540	3007700	1660080	6587320	6.916
1.000	143617	56977	84065	90153	15387	2.521	1919760	3059740	1577040	6556540	6.895
1.250	143617	41926	71586	84391	10617	3.425	1924460	3236740	1171728	6332928	6.151
1.500	143617	38555	67616	82314	9743	3.725	1917680	3297500	1077840	6293020	6.106
2.000	143617	31233	59408	78701	7751	4.598	1918660	3438840	875408	6232908	5.817
3.000	143617	17061	42334	70690	4050	8.418	1909800	3733340	481120	6124260	5.218
4.000	143617	8857	28822	60942	1946	16.215	1825600	3975020	252256	6052876	4.992
5.000	143617	5233	20854	53480	1465	27.445	1742340	4114020	143984	6000344	5.591

Data for F for varying ε											
ε	**No. TN**	**No. C**	**No. BT**	**No. G**	**No. B**	**Nodes / C**	**BT space**	**G space**	**B space**	**Total space**	**Cons. time**
0.250	4815484	83291	241944	242018	33327	57.815	4842080	96324560	2132048	103298688	50.050
0.500	4815484	81226	236337	237251	32428	59.285	4765940	96380920	2080352	103227212	49.525
0.750	4815484	72149	222171	226979	28522	66.744	4639560	96533300	1852384	103025244	49.113
1.000	4815484	71531	220324	225523	28112	67.320	4620120	96560600	1839168	103019888	49.154
1.250	4815484	42117	186608	202720	18322	114.336	4388160	96858460	1054560	102301180	48.307
1.500	4815484	41618	185050	201519	17990	115.707	4371400	96883380	1043904	102298684	48.382
2.000	4815484	33136	166501	188366	14401	145.325	4226600	97103080	829904	102159584	47.746
3.000	4815484	16887	124028	158258	8065	285.159	3800280	97737960	411312	101949552	66.508
4.000	4815484	45657	189612	204329	25450	105.471	4433800	97259140	1053792	102746732	48.256
5.000	4815484	101	51611	102157	39	47678.100	2735900	101210580	2576	103949056	45.994

Data for A1 for varying % of giraffe neck											
%	**No. TN**	**No. C**	**No. BT**	**No. G**	**No. B**	**Nodes / C**	**BT space**	**G space**	**B space**	**Total space**	**Cons. time**
0.200	3014010	186	930	2419	71	16204.400	172460	73768260	4784	73945504	127.403
0.300	3014010	186	930	2427	71	16204.400	172460	73831400	4784	74008644	127.286
0.400	3014010	186	930	2427	71	16204.400	172460	73823400	4784	74000644	127.286
0.500	3014010	186	930	4099	71	16204.400	172460	105646220	4784	105823464	136.896
0.600	3014010	186	930	5765	71	16204.400	172460	137179320	4784	137356564	146.647

Data for A2 for varying % of giraffe neck											
%	**No. TN**	**No. C**	**No. BT**	**No. G**	**No. B**	**Nodes / C**	**BT space**	**G space**	**B space**	**Total space**	**Cons. time**
0.200	3379858	279	1395	3632	107	12114.200	257400	79854260	7184	80118844	111.638
0.300	3379858	279	1395	3632	107	12114.200	257400	79764260	7184	80028844	113.055
0.400	3379858	279	1395	3647	107	12114.200	257400	80227800	7184	80492384	111.787
0.500	3379858	279	1395	6148	107	12114.200	257400	112361600	7184	112626184	121.498
0.600	3379858	279	1395	8648	107	12114.200	257400	144560360	7184	144824944	131.188

Data for A3 for varying % of giraffe neck											
%	**No. TN**	**No. C**	**No. BT**	**No. G**	**No. B**	**Nodes / C**	**BT space**	**G space**	**B space**	**Total space**	**Cons. time**
0.200	3377396	387	2610	4844	140	8727.120	358780	76363340	10112	76732232	79.134
0.300	3377396	387	2610	4860	140	8727.120	358780	76623660	10112	76992552	79.738
0.400	3377396	387	2610	8196	140	8727.120	358780	104676840	10112	105045732	88.172
0.500	3377396	387	2610	11517	140	8727.120	358780	132504960	10112	132873852	96.829
0.600	3377396	387	2610	11529	140	8727.120	358780	132713560	10112	133082452	96.913

Data for A4 for varying % of giraffe neck											
%	**No. TN**	**No. C**	**No. BT**	**No. G**	**No. B**	**Nodes / C**	**BT space**	**G space**	**B space**	**Total space**	**Cons. time**
0.200	3281877	480	3261	6056	171	6837.240	448520	71018460	12592	71479572	56.972
0.300	3281877	480	3261	6074	171	6837.240	448520	71210180	12592	71671292	57.283
0.400	3281877	480	3261	10710	171	6837.240	448520	95258360	12592	95719472	64.731
0.500	3281877	480	3261	15282	171	6837.240	448520	118989880	12592	119450992	71.577
0.600	3281877	480	3261	15349	171	6837.240	448520	119255420	12592	119716532	71.730

Data for A5 for varying % of giraffe neck											
%	**No. TN**	**No. C**	**No. BT**	**No. G**	**No. B**	**Nodes / C**	**BT space**	**G space**	**B space**	**Total space**	**Cons. time**
0.200	2812226	586	7258	7355	215	4799.020	604980	56776920	15280	57397180	42.343
0.300	2812226	586	7258	13425	215	4799.020	604980	71035440	15280	71655700	46.599
0.400	2812226	586	7258	18425	215	4799.020	604980	82840660	15280	83460920	50.278
0.500	2812226	586	7258	18430	215	4799.020	604980	82865060	15280	83485320	49.951
0.600	2812226	586	7258	18491	215	4799.020	604980	83469040	15280	84089300	49.818

Data for B1 for varying % of giraffe neck											
%	**No. TN**	**No. C**	**No. BT**	**No. G**	**No. B**	**Nodes / C**	**BT space**	**G space**	**B space**	**Total space**	**Cons. time**
0.200	2019486	1849078	1849078	1849078	618172	1.092	36981560	40389740	49279712	126651012	120.851
0.300	2019486	1849078	1849078	1849078	618172	1.092	36981560	40389740	49279712	126651012	120.816
0.400	2019486	1849078	1849078	1849078	618172	1.092	36981560	40389740	49279712	126651012	120.935
0.500	2019486	1849078	1849078	1849078	618172	1.092	36981560	40389740	49279712	126651012	122.423
0.600	2019486	1849078	1849078	1849078	618172	1.092	36981560	40389740	49279712	126651012	120.873

Data for B2 for varying % of giraffe neck											
%	**No. TN**	**No. C**	**No. BT**	**No. G**	**No. B**	**Nodes / C**	**BT space**	**G space**	**B space**	**Total space**	**Cons. time**
0.200	3078176	1139111	1562486	1562486	390508	2.702	31249720	61563540	30203392	123016652	101.594
0.300	3078176	1139111	1562486	1562486	390508	2.702	31249720	61563540	30203392	123016652	101.481
0.400	3078176	1139111	1562486	1562486	390508	2.702	31249720	61563540	30203392	123016652	101.970
0.500	3078176	1139111	1562486	1562486	390508	2.702	31249720	61563540	30203392	123016652	101.642
0.600	3078176	1139111	1562486	1562486	390508	2.702	31249720	61563540	30203392	123016652	101.534

Data for B3 for varying % of giraffe neck											
%	**No. TN**	**No. C**	**No. BT**	**No. G**	**No. B**	**Nodes / C**	**BT space**	**G space**	**B space**	**Total space**	**Cons. time**
0.200	2286364	757859	1503757	1503765	259629	3.017	30109920	45744520	20097392	95951832	76.829
0.300	2286364	757859	1503757	1504617	259629	3.017	30109920	45761560	20097392	95968872	77.471
0.400	2286364	757859	1503757	1504630	259629	3.017	30109920	45761820	20097392	95969132	77.078
0.500	2286364	757859	1503757	1504630	259629	3.017	30109920	45761820	20097392	95969132	76.968
0.600	2286364	757859	1503757	1504630	259629	3.017	30109920	45761820	20097392	95969132	77.119

Data for B4 for varying % of giraffe neck											
%	**No. TN**	**No. C**	**No. BT**	**No. G**	**No. B**	**Nodes / C**	**BT space**	**G space**	**B space**	**Total space**	**Cons. time**
0.200	2753553	588747	1251119	1284509	203821	4.677	26059220	56079920	15578736	97717876	73.418
0.300	2753553	588747	1251119	1285197	203821	4.677	26059220	56107440	15578736	97745396	73.332
0.400	2753553	588747	1251119	1285197	203821	4.677	26059220	56107440	15578736	97745396	73.452
0.500	2753553	588747	1251119	1285208	203821	4.677	26059220	56108100	15578736	97746056	73.417
0.600	2753553	588747	1251119	1285208	203821	4.677	26059220	56108100	15578736	97746056	73.302

Data for B5 for varying % of giraffe neck											
%	**No. TN**	**No. C**	**No. BT**	**No. G**	**No. B**	**Nodes / C**	**BT space**	**G space**	**B space**	**Total space**	**Cons. time**
0.200	3195515	517990	1538207	1568547	179269	6.169	31706140	64525280	13707344	109938764	78.617
0.300	3195515	517990	1538207	1569175	179269	6.169	31706140	64550400	13707344	109963884	78.501
0.400	3195515	517990	1538207	1569184	179269	6.169	31706140	64550940	13707344	109964424	78.556
0.500	3195515	517990	1538207	1569184	179269	6.169	31706140	64550940	13707344	109964424	78.514
0.600	3195515	517990	1538207	1569184	179269	6.169	31706140	64550940	13707344	109964424	78.147

Data for C1 for varying % of giraffe neck											
%	**No. TN**	**No. C**	**No. BT**	**No. G**	**No. B**	**Nodes / C**	**BT space**	**G space**	**B space**	**Total space**	**Cons. time**
0.200	2811774	18458	67276	67276	4591	152.334	1345520	56235500	517168	58098188	46.004
0.300	2811774	18458	67276	67276	4591	152.334	1345520	56235500	517168	58098188	45.935
0.400	2811774	18458	67276	67276	4591	152.334	1345520	56235500	517168	58098188	46.251
0.500	2811774	18458	67276	67276	4591	152.334	1345520	56235500	517168	58098188	45.854
0.600	2811774	18458	67276	67276	4591	152.334	1345520	56235500	517168	58098188	46.133

Data for C2 for varying % of giraffe neck											
%	**No. TN**	**No. C**	**No. BT**	**No. G**	**No. B**	**Nodes / C**	**BT space**	**G space**	**B space**	**Total space**	**Cons. time**
0.200	2825918	19269	68033	68033	5030	146.656	1360660	56518380	536096	58415136	46.189
0.300	2825918	19269	68033	68033	5030	146.656	1360660	56518380	536096	58415136	46.313
0.400	2825918	19269	68033	68033	5030	146.656	1360660	56518380	536096	58415136	46.184
0.500	2825918	19269	68033	68033	5030	146.656	1360660	56518380	536096	58415136	46.925
0.600	2825918	19269	68033	68033	5030	146.656	1360660	56518380	536096	58415136	45.943

Data for C3 for varying % of giraffe neck											
%	**No. TN**	**No. C**	**No. BT**	**No. G**	**No. B**	**Nodes / C**	**BT space**	**G space**	**B space**	**Total space**	**Cons. time**
0.200	2844036	19319	68171	68171	5080	147.214	1363420	56880740	536896	58781056	46.690
0.300	2844036	19319	68171	68171	5080	147.214	1363420	56880740	536896	58781056	46.342
0.400	2844036	19319	68171	68171	5080	147.214	1363420	56880740	536896	58781056	46.769
0.500	2844036	19319	68171	68171	5080	147.214	1363420	56880740	536896	58781056	46.499
0.600	2844036	19319	68171	68171	5080	147.214	1363420	56880740	536896	58781056	46.510

Data for C4 for varying % of giraffe neck											
%	**No. TN**	**No. C**	**No. BT**	**No. G**	**No. B**	**Nodes / C**	**BT space**	**G space**	**B space**	**Total space**	**Cons. time**
0.200	2817974	19492	68291	68291	5260	144.571	1365820	56359500	539552	58264872	46.189
0.300	2817974	19492	68291	68291	5260	144.571	1365820	56359500	539552	58264872	46.484
0.400	2817974	19492	68291	68291	5260	144.571	1365820	56359500	539552	58264872	46.505
0.500	2817974	19492	68291	68291	5260	144.571	1365820	56359500	539552	58264872	46.241
0.600	2817974	19492	68291	68291	5260	144.571	1365820	56359500	539552	58264872	46.229

Data for C5 for varying % of giraffe neck											
%	**No. TN**	**No. C**	**No. BT**	**No. G**	**No. B**	**Nodes / C**	**BT space**	**G space**	**B space**	**Total space**	**Cons. time**
0.200	2816441	20002	68724	68724	5412	140.808	1374480	56328840	553440	58256760	46.187
0.300	2816441	20002	68724	68724	5412	140.808	1374480	56328840	553440	58256760	46.122
0.400	2816441	20002	68724	68724	5412	140.808	1374480	56328840	553440	58256760	46.477
0.500	2816441	20002	68724	68724	5412	140.808	1374480	56328840	553440	58256760	45.887
0.600	2816441	20002	68724	68724	5412	140.808	1374480	56328840	553440	58256760	46.480

Data for D1 for varying % of giraffe neck											
%	**No. TN**	**No. C**	**No. BT**	**No. G**	**No. B**	**Nodes / C**	**BT space**	**G space**	**B space**	**Total space**	**Cons. time**
0.200	254051	123602	123717	123726	123541	2.055	2474700	5102640	1978576	9555916	1134.150
0.300	254051	123602	123717	123726	123541	2.055	2474700	5102640	1978576	9555916	1137.830
0.400	254051	123602	123717	123726	123541	2.055	2474700	5102640	1978576	9555916	1131.790
0.500	254051	123602	123717	123726	123541	2.055	2474700	5102640	1978576	9555916	1137.580
0.600	254051	123602	123717	123726	123541	2.055	2474700	5102640	1978576	9555916	1134.100

Data for D2 for varying % of giraffe neck											
%	**No. TN**	**No. C**	**No. BT**	**No. G**	**No. B**	**Nodes / C**	**BT space**	**G space**	**B space**	**Total space**	**Cons. time**
0.200	254550	123619	123767	123789	123527	2.059	2476220	5250000	1979344	9705564	700.659
0.300	254550	123619	123767	123789	123527	2.059	2476220	5250000	1979344	9705564	701.418
0.400	254550	123619	123767	123789	123527	2.059	2476220	5250000	1979344	9705564	701.042
0.500	254550	123619	123767	123789	123527	2.059	2476220	5250000	1979344	9705564	699.617
0.600	254550	123619	123767	123789	123527	2.059	2476220	5250000	1979344	9705564	704.443

Data for D3 for varying % of giraffe neck											
%	**No. TN**	**No. C**	**No. BT**	**No. G**	**No. B**	**Nodes / C**	**BT space**	**G space**	**B space**	**Total space**	**Cons. time**
0.200	255018	123612	123790	123823	123485	2.063	2477120	5501180	1979792	9958092	555.324
0.300	255018	123612	123790	123823	123485	2.063	2477120	5501180	1979792	9958092	558.495
0.400	255018	123612	123790	123823	123485	2.063	2477120	5501180	1979792	9958092	559.176
0.500	255018	123612	123790	123823	123485	2.063	2477120	5501180	1979792	9958092	558.465
0.600	255018	123612	123790	123823	123485	2.063	2477120	5501180	1979792	9958092	556.699

Data for D4 for varying % of giraffe neck											
%	**No. TN**	**No. C**	**No. BT**	**No. G**	**No. B**	**Nodes / C**	**BT space**	**G space**	**B space**	**Total space**	**Cons. time**
0.200	255450	123564	123781	123815	123407	2.067	2477340	5600440	1979504	10057284	509.520
0.300	255450	123564	123781	123824	123407	2.067	2477340	5852940	1979504	10309784	507.377
0.400	255450	123564	123781	123824	123407	2.067	2477340	5852940	1979504	10309784	511.838
0.500	255450	123564	123781	123824	123407	2.067	2477340	5852940	1979504	10309784	511.563
0.600	255450	123564	123781	123824	123407	2.067	2477340	5852940	1979504	10309784	511.978

Data for D5 for varying % of giraffe neck											
%	**No. TN**	**No. C**	**No. BT**	**No. G**	**No. B**	**Nodes / C**	**BT space**	**G space**	**B space**	**Total space**	**Cons. time**
0.200	256019	123652	123910	123941	123466	2.070	2480320	5613760	1981376	10075456	517.488
0.300	256019	123652	123910	123953	123466	2.070	2480320	5945840	1981376	10407536	517.068
0.400	256019	123652	123910	123962	123466	2.070	2480320	6263180	1981376	10724876	522.198
0.500	256019	123652	123910	123963	123466	2.070	2480320	6307260	1981376	10768956	523.088
0.600	256019	123652	123910	123963	123466	2.070	2480320	6307260	1981376	10768956	519.987

Data for E for varying % of giraffe neck											
%	**No. TN**	**No. C**	**No. BT**	**No. G**	**No. B**	**Nodes / C**	**BT space**	**G space**	**B space**	**Total space**	**Cons. time**
0.200	143617	67815	92998	93107	19160	2.118	1898840	2884560	1863488	6646888	7.226
0.300	143617	67815	92998	93301	19160	2.118	1898840	2888740	1863488	6651068	7.154
0.400	143617	67815	92998	93763	19160	2.118	1898840	2898200	1863488	6660528	7.369
0.500	143617	67815	92998	93791	19160	2.118	1898840	2899520	1863488	6661848	7.276
0.600	143617	67815	92998	93903	19160	2.118	1898840	2904460	1863488	6666788	7.225

Data for F for varying % of giraffe neck											
%	**No. TN**	**No. C**	**No. BT**	**No. G**	**No. B**	**Nodes / C**	**BT space**	**G space**	**B space**	**Total space**	**Cons. time**
0.200	4815484	81226	236337	237098	32428	59.285	4765940	96355580	2080352	103201872	49.499
0.300	4815484	81226	236337	237188	32428	59.285	4765940	96364860	2080352	103211152	49.553
0.400	4815484	81226	236337	237223	32428	59.285	4765940	96374360	2080352	103220652	49.457
0.500	4815484	81226	236337	237251	32428	59.285	4765940	96380920	2080352	103227212	49.508
0.600	4815484	81226	236337	237272	32428	59.285	4765940	96386980	2080352	103233272	49.520

References

[Aggarwal and Jeffrey, 1988] Alok Aggarwal and S. Vitter Jeffrey. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):pages **1116, 1117**, 1988.

[Brodal and Fagerberg, 2006] Gerth Stølting Brodal and Rolf Fagerberg. Cache-oblivious string dictionaries. In *Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages **581–590**, 2006.

[Cormen *et al.*, 2003] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, pages **21, 22, 269, 357, 358, 385**. The MIT Press, Cambridge, Massachusetts London, England, second edition, 2003.

[Dongarra *et al.*, 2003] Jack Dongarra, Phil Mucci, Dan Terpstra, Shirley Moore, Keith Seymour, and Haihang You. Performance application programming interface (papi), 2003.

[Farach-Colton *et al.*, 2000] Martin Farach-Colton, Paolo Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *J. ACM*, 47(6):pages **987, 988**, 2000.

[Ferragina and Grossi, 1999] Paolo Ferragina and Roberto Grossi. The string b-tree: a new data structure for string search in external memory and its applications. *J. ACM*, 46(2):pages **256, 257**, 1999.

[Fredkin, 1960] Edward Fredkin. Trie memory. *Commun. ACM*, 3(9):pages **490, 491**, 1960.

[Frigo *et al.*, 1999] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Proc. 40th Annual Symposium on Foundations of Computer Science*, pages **1, 2**. IEEE Computer Society Press, 1999.

[Huffman, 1952] David A. Huffman. A method for the construction of minimum redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40(9):page **1099**, Sep 1952.

[Morrison, 1968] Donald R. Morrison. Patricia-practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):pages **517–520**, 1968.

[Pan *et al.*, 2007] Shen Pan, Cary Cherng, Kevin Dick, and Richard E. Ladner. Algorithms to take advantage of hardware prefetching. In *Proc. 9th Workshop on Algorithm Engineering and Experiments*, page **1**, 2007.

[van Emde Boas *et al.*, 1977] Peter van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:page **106**, 1977.

[von Neumann, 1945] John von Neumann. First draft of a report on the edvac. pages **1–3**, 1945.

[Wall, 2006] Larry Wall. Practical extraction and report language (perl), 2006.

[Williams and Kelley, 2004] Thomas Williams and Colin Kelley. Gnuplot, 2004.

Index