Karlsruhe Institute of Technology

Engineering Succinct Predecessor Data Structures

Master Thesis of

Jan Benedikt Schwarz

at the Department of Informatics
Institute of Theoretical Informatics, Algorithm Engineering

Reviewer:	Prof. Dr. Peter Sanders
Second reviewer:	
Advisor:	M.Sc. Hand-Peter Lehmann
Second advisor:	Dr. Florian Kurpicz

12.01.2023 – 12.07.2023

I declare that I have developed and written the enclosed thesis completely by myself. I have submitted neither parts of nor the complete thesis as an examination elsewhere. I have not used any other than the aids that I have mentioned. I have marked all parts of the thesis that I have included from referenced literature, either in their original wording or paraphrasing their contents. This also applies to figures, sketches, images and similar depictions, as well as sources from the internet.

Ort: Karlsruhe, Datum:

. . . . . .

(Jan Benedikt Schwarz)

Abstract

We implement and modify the predecessor data structure of Sarel Cohen et al. to allow predecessor queries using $O(\log_{w/logw} n)$ probes with no restrictions on n in $O(\log n)$ time using 1% additional memory. These modifications include building a tree with γ-Nodes as inner nodes. Furthermore, we implement a portion of the predecessor data structure described in the attachment of "Optimal lower and upper bounds for representing sequences" by Belazzougui et al. and provide insight on the problems of the original definition.

To evaluate our implementations we conduct experiments with different values for w and compare their performance with other, more established, predecessor data structures. Our experiments show that for the currently common 64-Bit architectures γ-Nodes are only beneficial if accessing data is a major time bottleneck.

Zusammenfassung

Wir implementieren und modifizieren die Vorgänger Datenstruktur von Sarel Cohen et al, was uns erlaubt Vorgängeranfragen mit $O(\log_{w/logw} n)$ Zugriffen auf den Speicher in $O(\log n)$ Zeit zu beantworten, ohne die maximale Anzahl an Elementen einschränken zu müssen. Dies wird durch den Aufbau eines Baumes mithilfe von γ-Knoten erreicht. Der für diese Datenstruktur erforderliche Speicher kann weniger als 1% des Speichers, der für die ursprünglichen Daten erfordert wird, betragen.

Des Weiteren werden Teile des Anhangs aus "Optimal lower and upper bounds for representing sequences" von Belazzougui et al. implementiert, sowie auf die Probleme dieser Datenstruktur eingegangen.

Um unsere Implementationen zu beurteilen wird mit verschiedenen Werten für w experimentiert und die Performance mit bereits etabliert Datenstrukturen verglichen. Unsere Experimente zeigen, dass γ-Knoten bei den momentan gängigen 64 Bit Betriebssystemen nur von Vorteil sind, wenn der Zugriff auf den Speicher zeitlich stark eingeschränkt ist.

Contents

1 Introduction

The predecessor problem is an easy to understand problem that involves finding the largest element in a set that is equal or less than a given query element. While binary search provides a straightforward solution, the problem is well researched as there are a lot of space-time trade-offs possible. Space consumption is an important aspect for any such data structure. If the additional space a data structure is allowed to allocate is sublinear in the space of the overall data, only a handful of data structures remain.

The current trend is to collect and store more and more data, resulting in an increased memory usage, which can become pricey. If the used index requires linear or worse space compared to the data, such costs would increase even more. Because of this, using an index with sublinear space is a great alternative. The logical correctness and space requirements of multiple such indices have been proven in theory. However, in practice, few have been implemented. It is therefore necessary to implement, test and compare more of these indices to allow for faster predecessor queries using less space.

The succinct data structure by Sarel et al.[3] can solve predecessor queries on a set of size $\frac{w}{\log_2 w}$ in $O(\log w)$ time using $O(1)$ probes. While impressive in theory, in practice 64-bit architectures are the most common and for $w = 64$ the data structure only supports up to 8 elements. This in turn makes the constant factor omitted by the big-O-notation far more important.

Our results show that a tree with nodes based on the structure by Sarel et al.[3] can be beneficial if a low number of data accesses is important. In this case, for 128 bit words, such a tree is twice as fast as binary search while using less than 1% additional memory. If accessing data is not a significant time constrain, the structure demonstrates much slower performances than its alternatives. We also show that the data structure in the attachment of "Optimal lower and upper bounds for representing sequences" [2] is highly situational as the first step in it can reduce the search range to a near constant factor if the distribution of the data is uniform, or inefficacious if the distribution is highly concentrated around a few values. Should the distribution be uniform the described rank select data structure on its own, paired with a binary search, performs remarkably while occupying little memory. The reduced asymptotic space of the van Emde Boas tree in the second step is often ineffective because of large constant factors if the universe is not densely populated.

2 Preliminaries

Given a universe $\mathcal{U}$, a set $S \subset \mathcal{U}$ and an element $e \in \mathcal{U}$, the predecessor of e in S is $\texttt{pred}(e) = \max\{x \in S | x \leq e\}$. For the static predecessor problem, S is provided at the start and does not change over the course of time. In the dynamic case, insert and delete operations have to be provided to allow for the addition and removal of elements in S during runtime.

The predecessor problem can be solved optimally in the comparison model by using a binary search tree, however it is more complex in RAM models, in which all elements of $\mathcal{U}$ can be represented using a binary string of finite length w. In these models the possible solutions can vary in speed, occupied space, usage of randomization or multiplication and more. Should space be disregarded, one can simply store the answer for every element in the finite universe of the model. If a data structure performs predecessor queries in $\Omega(\log |S|)$ time, in most cases, it can be replaced with a binary search to reduce the memory consumption.

In this thesis we assume that a universe $\mathcal{U}$ is a collection of integer numbers starting at 0 and ending at $|\mathcal{U}| - 1$. The word size $w = \lceil \log_2 |\mathcal{U}| \rceil$ indicates how many bits are needed to represent the elements of $\mathcal{U}$. Given $x \in \mathcal{U}$, the bit of x at index i is x_i, with x_0 as the least significant bit and x_{w-1} as the most significant bit. Given a list of indices I, $x[I]$ is a element that uses $|I|$ bits with $x[I]_i = x_{I[|I|-i]}$.

In general, the used memory can be split into two categories. In the first category are the bits necessary to store the data itself without compression $B(|S|, |\mathcal{U}|) = \lceil \log_2 \binom{|S|}{|\mathcal{U}|} \rceil$. The additional memory $R(|S|, |\mathcal{U}|)$ needed for the index will be labeled the redundant part. If used correctly, $R(|S|, |\mathcal{U}|)$ can be used to reduce the time the index needs for operations. An index with $R(|S|, |\mathcal{U}|) \in o(B(|S|, |\mathcal{U}|))$ is called succinct.

Multiple lower bounds for $R(|S|, |\mathcal{U}|)$ in relation to the time predecessor queries take are shown by Mihai Pătraşcu and Mikkel Thorup in “Time-space trade-offs for predecessor search” [13].

The predecessor problem can also be reduced to a rank and select operation by storing S as a bit array a of length $|\mathcal{U}|$ where $a_i = 1 \iff i \in S$. A rank select data structure provides two operations. $\texttt{rank}_b(a, x)$ is the number of bits set to $b \in \{0, 1\}$ in a up to index i and $\texttt{select}_b(a, x)$ is the index of the xth bit set to b in a. The predecessor of x is $\texttt{pred}(e) = \texttt{select}_1(a, \texttt{rank}_1(a, e) - 1)$. While $\texttt{select}_b(a, x)$ and $\texttt{rank}_b(a, x)$ can be computed in constant time, in many cases S is much smaller than $\mathcal{U}$ while a requires $\mathcal{U}$ bits, making them impractical for predecessor searches unless the bit array is compressed [11].

3 Related Work

3.1 Fusion Trees

One data structure that solves the predecessor problem is the fusion tree [7, 6]. Fusion trees are very similar to B-Trees, but they have a branching factor of $k = w^e | e \in (0, 1)$. A node in the fusion tree uses one word for every child it has to determine in which child a predecessor search should continue. Because of the high branching factor, a specially designed compression called sketching has to be used on each word to find the correct child in constant time.

To create the sketch, first define a set S of k words. S is responsible for one node of the fusion tree and contains one word for every child the node has. The words in S are then combined in a trie. Such a trie will have k branches, with branches occurring at the levels $b_1, b_2, ...b_r | r \leq k$ of the trie. The perfect sketch of a word w can be build using $\texttt{sketch}(w)_i = w_{b_i} | i \in [1, r]$, with each sketch only using r bits. It is possible to distinguish all words in S from each other by only using their sketches. In addition, sketches preserve order inside the set of words used to build them $\forall p, q \in S : \texttt{sketch}(p) < \texttt{sketch}(q) \iff p < q$.

The difficulty of creating sketches is that there is no obvious way for extracting the bits $w_{b_1}, w_{b_2}, ...w_{b_r}$ and storing them in a compact block in $O(1)$ time. A non-perfect sketch is a sketch that contains more than r bits, these additional bits are not used for anything but can allow a faster calculation of a sketch. Fredman and Willard showed a method for non-perfect sketches in the special case of $k = w^{\frac{1}{5}}$. They first prove that for any combination of the integers $b_1, b_2, ...b_r$, a sequence of integers $m_1, m_2, ...m_r$ exists such that:

$$m_1 + b_1 < m_2 + b_2 ... < m_r + b_r, \tag{3.1}$$

$$m_i + b_j \text{ is distinct } \forall i, j \in [1, r] \tag{3.2}$$

$$(m_r + b_r) - (m_1 + b_1) = O(w^{\frac{4}{5}}) \tag{3.3}$$

Calculating this sequence is slow. However, it only has to be calculated once, when the fusion tree is built.

To create a sketch of w, the unimportant bits from w are first removed using an AND operation with $\Sigma_{i=1}^{r} 2^{b_i}$, the result is multiplied with $\Sigma_{i=1}^{r} 2^{m_i}$. This shifts the important bits closer together. To remove the unwanted bits created by the multiplication, another AND operation, this time with $\Sigma_{i=1}^{r} 2^{m_i + b_i}$, is needed. Finally, everything is shifted to the right by $m_1 + b_1$ bits and the sketch is the first $m_r + b_r - m_1 + b_1$ bits.

A node u in a fusion tree stores the bit arrays $\Sigma_{i=1}^{r} 2^{m_i}$ and $\Sigma_{i=1}^{r} 2^{m_i + b_i}$ to create a sketch for any word it is given. In addition, the sketch of every element $S[i]$ in S is stored together as one sequence $\texttt{sketch}(u)$, separating each $\texttt{sketch}(S[i])$ is a bit set to one.

$$\texttt{sketch}(u) = 1, \texttt{sketch}(S[0]), 1, \texttt{sketch}(S[1]), ..., 1, \texttt{sketch}(S[k-1]) \tag{3.4}$$

Because a sketch has length $O(w^{\frac{4}{5}})$ bits and there are $O(w^{\frac{1}{5}})$ sketches, $\texttt{sketch}(u)$ has a total length of $O(w)$ bits. Given word q and the number of bits a sketch of an element occupies d, a node does the following calculation to compare the sketch of every child of u with q in $O(1)$:

$$\texttt{diff} = (\texttt{sketch}(u) - \texttt{sketch}(q) \cdot \Sigma_{i=0}^{w^{\frac{1}{5}}-1} 2^{i(d+1)}) \text{ AND } \Sigma_{i=1}^{w^{\frac{1}{5}}} 2^{i(d+1)} \tag{3.5}$$

Note that the two sums are part of the preprocessing and not calculated at runtime. For any any index $i \in [0, |S|)$, $\texttt{sketch}(S[i]) < \texttt{sketch}(q)$ implies that $\texttt{diff}$ at index $(|S| - i) \cdot (d + 1)$ is zero. Because of the mask applied to diff, the first child with a sketch bigger or equal the sketch of q can be found by searching the most significant bit with a value of one in $\texttt{diff}$.

To find the predecessor of an element x in a fusion node one first searches the predecessor of $\texttt{sketch}(x)$. Using parallel comparison, one finds an index i for which $\texttt{sketch}(S[i]) \leq \texttt{sketch}(x) \leq \texttt{sketch}(S[i+1])$ in constant time. Let y be either $S[i]$ or $S[i+1]$, whichever shares a longer prefix with x. y is not guaranteed to be the predecessor of x, but it shares the longest common prefix (LCP) with x out of all the elements in S. To find the predecessor a second predecessor query on the sketches has to be done. Define j to be the length of the LCP between x and y and use it to form a new element r:

$$r = x_0, x_1 ..., x_{j-1} + \begin{cases} 0, 1^{w-j-1}, & \texttt{if } x_j = 1 \\ 1, 0^{w-j-1}, & \text{otherwise} \end{cases} \tag{3.6}$$

Using parallel comparison one can now find an index j such that $\texttt{sketch}(S[j]) \leq \texttt{sketch}(r) \leq \texttt{sketch}(S[j+1])$. The predecessor of x is either $S[j]$ or $S[j+1]$.

The first step of building a fusion tree on a sorted set S' of any size is the creation of fusion nodes on every k consecutive elements of S' to create the leaves of the tree. In the second step the smallest element of every fusion node from the previous step is used to created the next level of the tree. This is repeated until one fusion node remains, which will be the root of the tree. To find the predecessor of q in S', one repeatedly searches for the child of a fusion node responsible for it, starting at the root.

For example, using $w = 20$ and $S = \{$ 01000000000000000110, 01000000000000000111, 01001000000001000000, 01001000000001000001, 01001000000001000010 $\}$, one can build the fusion node depicted in Figure 3.1. The sketch of a word only contains the bits at index 15, 1 and 0. Thus the perfect sketches of the elements in S are 010, 011, 100, 101 and 110. The fusion node would therefore store the word $\texttt{sketch}(S) = 10101011110011011110$. When searching the predecessor of e.g. $e = 01100000100001000111$, first $\texttt{sketch}(e) = 011$ is computed. The sketch of e is then multiplied $\texttt{sketch}(e) \cdot 00010001000100010001 = 00110011001100110011$ and subtracted from $\texttt{sketch}(S)$, resulting in the word 01101000100110101011. After applying the mask one gets 00001000100010001000. The first bit set to one is at index 15, which is responsible for $S[1]$. $S[1]$ and $S[1-1]$ share the same LCP of length 2 with e. Because the third most significant bit of e is one, q is 01011111111111111111. After making a sketch of q, multiplying it and subtracting it from $\texttt{sketch}(S)$, one finds out that after applying the mask all bits are set to zero. Therefore all elements are smaller than e and the biggest element in S is the predecessor.

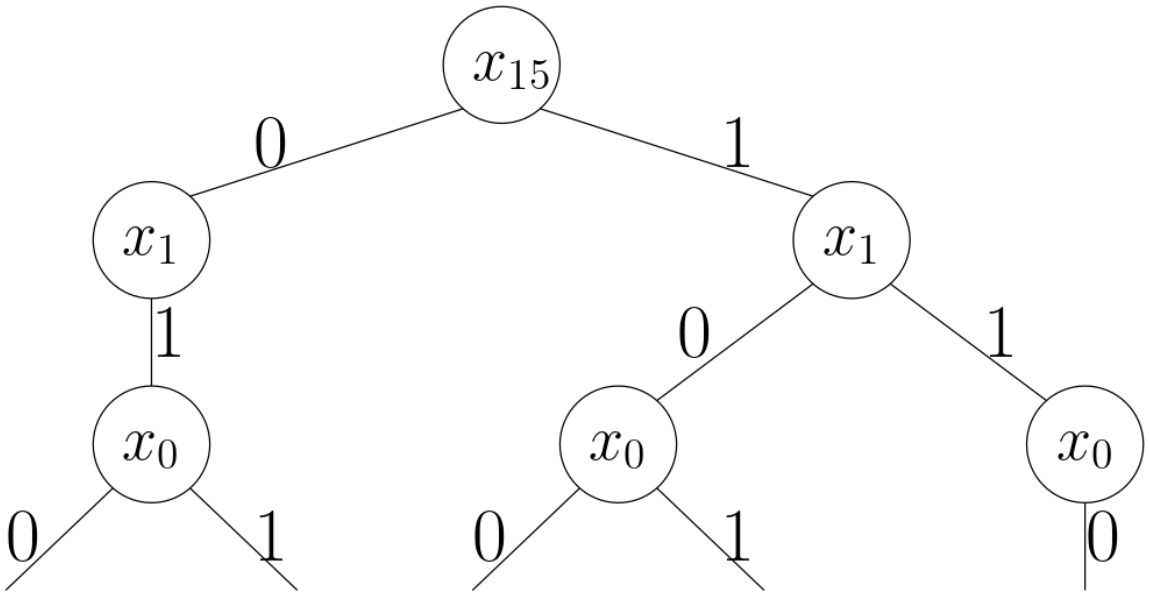


Figure 3.1: Example for the tree of a fusion node with
$S = \{$ 0100000000000000110, 0100000000000000111, 0100100000001000000, 0100100000001000001, 0100100000001000010$\}$.

3.2 Van Embde Boas Trees

Van Embde Boas (vEB) Trees were introduced by Peter van Embde Boas in his paper "Preserving order in a forest in less than logarithmic time" [4] in 1975.

Given a set $S \in \mathcal{U}$, the root of the vEB tree splits every element x in S into a `high` and `low` part. `high` contains the $\frac{w}{2}$ most significant bits $\texttt{high}(x) = x_{w-1}, x_{w-2}, ..., x_{w/2}$ and `low` the $\frac{w}{2}$ least significant bits $\texttt{low}(x) = x_{w/2-1}, x_{w/2-2}, ..., x_0$. Elements with the same `high` part form clusters together. For every cluster the root has a child which has universe $[0, 2^{w/2})$ and contains the `low` part of all elements in the cluster. In addition, the root has one more child, called the summary. The summary contains all the different `high` parts as elements and also operates on the universe $[0, 2^{w/2})$. All children of the root repeat this process of splitting its elements in half on their smaller universe, recursively halving the element size in every iteration. The predecessor of an element e is either in the cluster responsible for $\texttt{high}(e)$ or is the biggest element in the cluster responsible for the predecessor of $\texttt{high}(e) - 1$, which can be found using the summary. A predecessor search first searches for the node responsible for $\texttt{high}(e)$ and, if said node exists and its smallest element is not bigger than e, continues by searching the predecessor of $\texttt{low}(e)$ inside it. Otherwise, it searches the predecessor of $\texttt{high}(e) - 1$ in its summary vector, this will return $\texttt{high}(\texttt{pred}(e))$. To find $\texttt{low}(\texttt{pred}(e))$ one searches the child responsible for $\texttt{high}(\texttt{pred}(e))$ for its biggest element. In order for the vEB tree to achieve $O(\log w)$ time, the nodes have to store the maximum and minimum element they contain directly. This allows the predecessor searches recursion to not split as it either continues in the child responsible for $\texttt{high}(x)$ or the summary, but never both.

There are several ways of finding the position of the child responsible for high(e). While pointers to all possible children can be stored directly in their parent node, for large universes

it is often preferred to use hashing to reduce the redundant space to linear [17]. If hashing is used, the `high` part of every non-empty cluster functions as key and the pointer to the node responsible for said cluster as value. This still allows access times in constant time while removing the need to reserve space for pointers to empty clusters [13].

The asymptotic time of $O(\log w)$ of the vEB tree can be further reduced to $O(\frac{\log w}{\log \log w})$ with the modifications by Beame and Fich [1], however, this requires quadratic space.

3.3 γ-Nodes

γ-Nodes were introduced by Sarel Cohen et al. [3] in 2015 as successors to α- and β-Nodes, which were also presented in the same paper. A γ-Node can solve predecessor queries by employing $O(1)$ probes in $O(\log w)$ time. However, its capacity is limited to only $O(\frac{w}{\log w})$ elements. To find the predecessor of an element x, the γ-Node first searches the element which shares the longest common prefix (LCP) with x amongst all elements within the γ-Node. This search is called a blind search (subsection 3.3.1) by Sarel Cohen et al. [3] and the element found by the blind search is noted as $bkey(x)$ and is positioned at index $bs(x)$. To determine $pred(x)$, an intermediate element z is required. Let b represent the bit of x immediately following the LCP it shares with $bkey(x)$. z is composed of the LCP shared by x and $bkey(x)$ in its most significant bits, followed by a repetition of b to fill all the remaining less significant bits.

The predecessor of x is either $bkey(z)$ if $bkey(z) \leq x$ or at index $bs(z) - 1$. If $bs(z) = 0$ and $bkey(z) > x$, no predecessor exists. One probe on the elements is required for calculating z, and another is required for comparing $bkey(z)$ with x. No other probes are required to find the index of the predecessor and therefore the total number of probes is constant. The blind search can be performed in $O(\log w)$ time and requires $O(w)$ space (subsection 3.3.1).

Since γ-Nodes have a limited capacity of up to $O(\frac{w}{\log w})$ elements, Sarel Cohen et al. [3] propose a strategy of partitioning the bigger sets of elements into blocks. Within each block, a γ-Node is utilized for predecessor searches. To determine the block that contains the predecessor, a different algorithm can be used. The suggested algorithms for this purpose include fusion trees [7], y-fast-tries [18] and the optimal structure proposed by Pătraşcu and Thorup[13].

3.3.1 Blind Search

Given a sorted set S of elements $y_0 < y_1 < ... < y_k$ with $k < \log_2(w)$ and an element x, a γ-Node first builds a blind trie using the elements of S. A blind trie is a tree with $|S|$ leaves, each representing one element in S. An internal node u of the trie contains the index i_u of the bit right after the LCP that all leaves below it share. u has two children, the leaves which are an ancestor of the left child of u have a zero at index i_u and the leaves which are an ancestor of the right child of u a one. Let $I_{q_{q\in[0,k]}}$ be the indices of the nodes on the path from the root of the trie to the leaf representing y_q. A blind search of x starts by searching the range $L = 0$ to $R = k$ and halves this range in every step by checking if x is smaller than the element at index $q = L + \lfloor \frac{R-L}{2} \rfloor$, the middle of said range. For this it compares $x[I_q]$ with $y_q[I_q]$. If $x[I_q]$ is greater, the range is reduced to $[L+q+1, R]$ and if it is less the range is reduced to $[L, L+q-1]$. This pattern is repeated until the two terms are equal, at which point the search ends and the current index is returned.

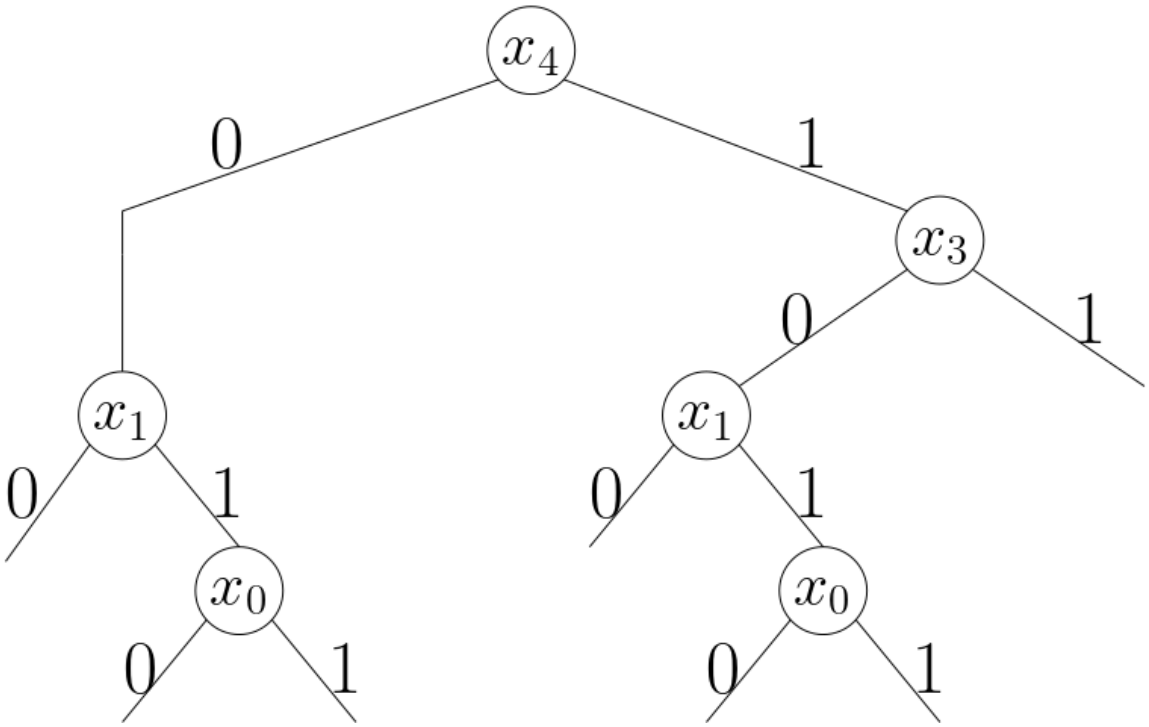


Figure 3.2: Example for a trie with $S = 00101, 00110, 00111, 10001, 10010, 10011, 11101$. The bits with lower opacity are irrelevant for the structure of the blind trie and not stored in it.

In each step the blind search has to compute $x[I_q]$ and $y_q[I_q]$ in constant time. This is achieved by splitting both into two parts. The first parts, composed of the most significant bits, can be taken from a previous step. At any point, the path from the root to q shares its first nodes with the path to L and R, but it shares longer path with either L or R. Without loss of generality, let q share more inner nodes with L. Both $x[I_L]$ and $y_L[I_L]$ have been used in a previous step, thus one can simply copy the first bits of both into $x[I_q]$ and $y_q[I_q]$, for the first iteration they are precomputed. To know how many bits q and L share, a array j_l is precomputed which stores said information for every element in the γ-Node. A similar array j_r which contains the length of the LCPs between the possible q and R combinations has to be stored as well. This also provides a convenient way of checking whether q shares a longer path with L or R in the trie, as $j_l[q]$ can be compared with $j_r[q]$.

To get the bits at the later indices $\tilde{I}_q$ of I_q, which are not shared with L and R, a different approach is needed for $x[\tilde{I}_q]$ and $y_q[\tilde{I}_q]$. For $y_q[I_q]$ the bits are computed for all elements in S during the preprocessing and stored in a bit array Z. To know where the bits of element q begins, a second array s is used which stores the prefix sum of the sizes of $\tilde{I}_q$, $s_q = \sum_{i=0}^{i \leq q} |\tilde{I}_i|$ and therefor $y_q[\tilde{I}_q] = Z[s_{q-1}, s_{q-1} + 1, ..., s_q - 1]$.

To get $x[\tilde{I}_q]$ a bit selector is used on x to create $X = x[\tilde{I}_0, \tilde{I}_1, ..., \tilde{I}_k]$. X is computed at the start of the blind search in time $O(\log w)$ using the bit selector by Sarel Cohen et al. [3] (subsection 3.3.3). To extract $x[\tilde{I}_q]$ from X the prefix sum array s is used.

By concatenating the bits known from the previous steps with $x[\tilde{I}_q]$ or $y_q[\tilde{I}_q]$ one can compute $x[I_q]$ and $y_q[I_q]$ to compute which of the two is bigger and reduce the search range accordingly.

An example of a trie is depicted in Figure 3.2. For S =00101, 00110, 00111, 10001, 10010, 10011, 11101 the indices along the left most path are $I(00101) = [4, 1]$ and $I(11101) = [4, 3]$ along the right most path. The first element that the blind search inspects is the median element $q = 100011$ with $I(q) = [4, 3, 1]$. $I(q)$ shares a longer prefix with $I(11101)$ than it does with $I(00101)$. Therefore, the prefix that $I(q)$ shares with $I(11101)$ is removed to calculate $\tilde{I}(q) = [1]$ and $y_q[\tilde{I}(q)] = 0$.

The bit selector, s and Z all require $O(w)$ space and $|S| \in O(\frac{w}{\log w})$. However, while implementing the above described γ-Node, we noticed that the constant factor obscured by the big-O-notation is larger than one and thus some values did not fit into one machine word. To fix this a slightly different method was used which still has the same time complexity but allows all operations to be handled in one machine word (see chapter 4).

3.3.2 Benes Networks

A Benes network is a communication network composed of a number of switches which have two inputs (x_1, x_2) and two outputs (y_1 and y_2) [10]. Each individual switch can either connect x_1 to y_1 and x_2 to y_2 (`direct`) or connect x_1 to y_2 and x_2 to y_1 (`crossed`).

In the context of Sarel Cohen et al. [3] and this work a Benes network is used to realize a permutation p on the bits of an element e such that $p(e)_i = e_{p_i}$.

Benes networks can be defined recursively. A 1-Benes network is a single switch with two inputs and outputs. A $(r + 1)$-Benes network has 2^{r+1} inputs and outputs, it consists of two layers of 2^r switches with two r-Benes networks (B_0, B_1) between them. In the first layer, for $q \in [0, 1]$ the output y_q of switch i is connected to the i'th input of B_q. Mirrored, the i'th output of B_q is the input x_q of the i'th switch in the last layer. In the first layer of a $r + 1$-Benes network the first input of switch s_i is e_i and the second input is e_{i+2^r}. In the last layer, the first output of a switch s_i is $p(e)_i$ and the second output is $p(e)_{i+2^r}$.

Any permutation of 2^r bits can be realized with a r-Benes network [10]. If $r \leq w/\log_2(w)$, each layer can be computed in parallel and the total time for a permutation is in $O(r)$ using $O(w)$ space [3]. The permutation of two bits can be realized by a 1-Benes network by using either a `crossed` or `direct` switch. The creation of a $(r + 1)$-Benes network for permutation p can be split into two repeating phases. In the first phase, a configuration for the switches of the first and last layer is computed such that a permutation for B_1 and B_2 exists which allows the two layers in conjunction with B_1 and B_2 to realize p. In the second phase, these permutations are calculated and used to compute the r-Benes networks B_1 and B_2.

A valid switch configuration can be found using the following method: Find a switch with no configuration in the left layer s_{l_i} and set it to `direct`, thus the bit e_i will be sent to B_0. Set the permutation of B_0 such that e_i is at output $p_i \mod 2^r$ thus e_i will be sent to the switch $s_{r_{p_i} \mod 2^r}$ in the last layer, which is responsible for out output p_i. The switch is set to either `direct` if $p_i < 2^r$ or `crossed` otherwise. Now we have to make sure that the second input of $s_{r_{p_i} \mod 2^r}$ is sent to B_1 by setting the configuration of another switch s_{l_j} in the first layer. The second input of $s_{r_{p_i} \mod 2^r}$ has to go through B_1 because it is impossible to connect it to the switch otherwise, as a switch in the last layer always gets one input from B_0, which it already

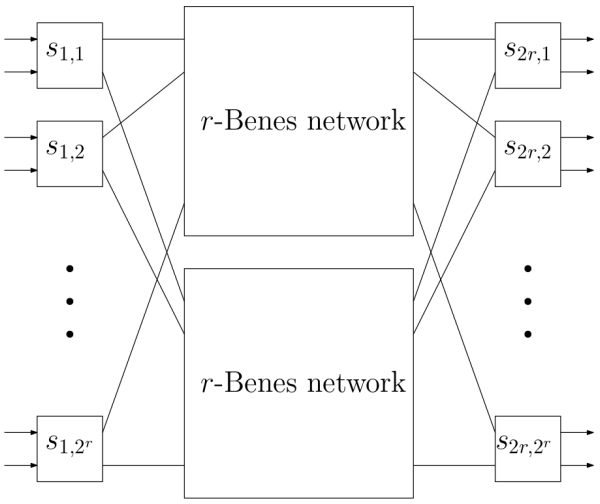


Figure 3.3: Contruction of a $(r+1)$-Benes network, source: [10]. The source was slightly edited, as the original implies that a r-Benes network always has four inputs and outputs.

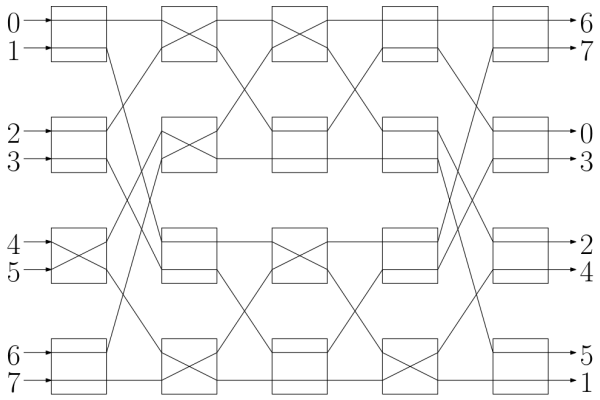


Figure 3.4: Example of a 3-Benes Network with the permutation [2, 7, 4, 3, 5, 6, 0, 1].

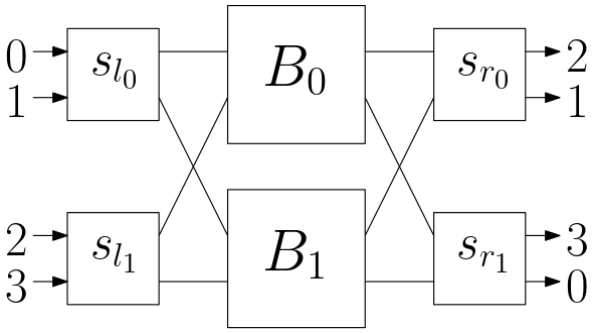
does, and one from B_1 (and there is no communication between B_0 and B_1). After this has been done, two possibilities arise. The first possibility is that s_{l_j} is actually the switch s_{l_i}, which the algorithm started with. In this case we either repeat the process with a new switch, assuming there are still switches with no configuration, or go into stage 2. In the case of $s_{l_j} \neq s_{l_i}$, it indicates that the other output of s_{l_j} now flows into B_0, and we must ensure that it ends up at the correct position. This can be achieved by repeating the previously described process. Once all switches in the first and last layer have a configuration, it is time to calculate the permutations of B_0 and B_1. Let $\tilde{S}$ be the set of indices of switches in the first layer that are set to `crossed`. We now define a new permutation $\tilde{p}$ for the two inner Benes-networks:

$$\tilde{p}_i = \begin{cases} p_i \mod 2^r & (i \mod 2^r) \notin \tilde{S} \\ p_{(i+2^r) \mod 2^r} \mod 2^r & (i \mod 2^r) \in \tilde{S} \end{cases} \tag{3.7}$$

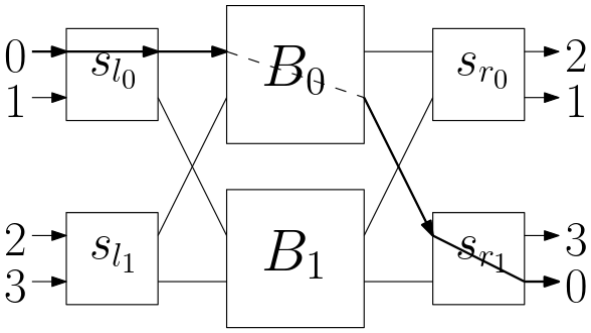
The permutation of B_0 is $\tilde{p}[0, 1, ..., 2^r - 1]$ and for B_1 it is $\tilde{p}[2^r, 2^r + 1, ..., 2^{r+1} - 1]$.

3.3.3 Bit Selector

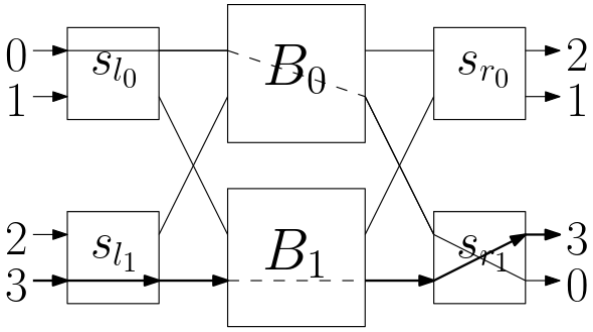
Given an element $x \in \mathcal{U}$ and a list L of indices $[i_0, i_1, ..., i_n] \mid i_j < w, \forall j \in [0, n]$, the elements composed of the bits in x at the indices in L is $x[L] = [x_{i_0}, x_{i_1}, ..., x_{i_n}]$. A bit selector is a algorithm that computes $x[L]$. The simplest bit selector algorithm extracts each bit individually, requiring $O(|L|)$ time. The bit selector introduced by Sarel Cohen et al. [3] only requires $O(|\log w|)$ time but requires $|L| \leq \frac{w}{\log_2 w}$. It makes multiple uses of Benes networks [10] to rearrange bits. The bit selector is split into 8 phases, with the first 5 phases being used to shift the selected bits to the front, the 6th and 7th to duplicate bits whose indices occur multiple times in L and the last phase to shift the bits to the correct position. The bits of x can be split into $w/\log_2(w)$



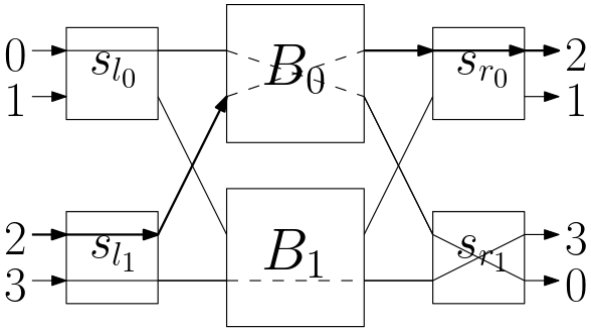
(a) Benes network with no configuration for the switches, the lables at the outputs on the right indicates the the input that should be connected to it.



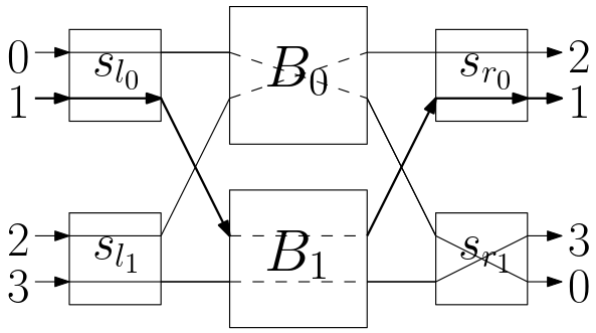
(b) In the first step s_{l_0} is set to `direct` and as a consequence s_{r_1} is set to `crossed`.



(c) Because s_{r_1} was set to `crossed`, the forth bit of the input has to be sent to B_1, therefore s_{l_1} is set to `direct`.



(d) Because s_{l_1} was set to `direct`, the third bit of the input is sent to B_0 and s_{r_0} has to be set to `direct`.



(e) s_{r_0} was set to `direct`, thus the first input has to be sent to B_1 by setting s_{l_0} to `direct`. This is already the case and the algorithm ends because all switches have a configuration.

Figure 3.5: Example for a 2-Benes network construction using $p = [2, 1, 0, 3]$.

consecutive blocks b_0 to $b_{\log_2(w)-1}$ with the length $\log_2(w)$. A bit x_i is part of block $b_{\lfloor i/\log_2(w)\rfloor}$.
Phase 0, Removing unimportant bits: For the bit selector, all bits whose index does not appear in L are unimportant and removed in the first step. To achieve this, a mask M is created whose bits are 1 at the indices in L and 0 everywhere else, $x' = x$ AND M.
Phase 1, Packing bits inside their block: All important bits are shifted to the left most positions inside their block while keeping their order. For this a loop with $\log(w)$ iterations is used. In each iteration i, the rightmost i bits of each block are shifted one step to the left if the bit there is not important, or stay the same if said bit is important. Each iteration of the loop can be computed in constant time.
Phase 2, Sorting blocks by their important bits count: All blocks are sorted in descending order by the number of important bits they contain. Note that a block is sorted by the amount of bits that are important and it does not matter if the index of a bit occurs multiple times in L. For the sorting, a modified Benes network (subsection 3.3.2) that works on blocks instead of bits is used. The Benes network can be computed during the initialization and only takes $O(\log w)$ time when a predecessor query is made.
Phase 3, Dispersing the bits: In this step every important bit is assigned to a single, distinct block of x and moved to the blocks left most position. The position of this block depends on the block the bit is currently in, the index it is at and how filled the other blocks are. Bits at the first index of their block are not moved, all bits at the second index of their block are shifted at the same time by the same amount of bits such that the bit in the first block will be in the empty block next to the first non-empty block. This is repeated for the bits at the third index of their block and so on.
Phase 4, Compacting the bits: The bits are moved to the left most positions. If there are n important bits they should occupy the bits at indices x_0 to x_{n-1} after this phase, thus occupy the left most $q = \lceil \frac{n}{\log_2(w)} \rceil$ blocks. First the $q-1$ left most blocks are filled in parallel in $O(\log w)$ time. They already start with a important bit in their left most bit. To fill the second left most bit, the blocks b_{q-1} to b_{q+q-3} are shifted to the left by $\log_2(w) \cdot q - 2$ bits and added to the first $q-1$ blocks. Now the two left most positions of the first $q-1$ blocks contain important. The process is repeated $\log_2(w) - 2$ times to fill the block entirely. Finally, to partially fill the last block b_{q-1} the bits of the remaining blocks, which did not get added to the previous blocks, are added individually to b_{q-1}.
Phase 5, Spacing the bits: Some bits have indices that appear multiple times in L and will be duplicated in phase 6. In this step we prepare for the duplication by creating empty spaces after bits that will be duplicated. This is achieved by a Benes network that takes the n important bits that are all positioned at very left, plus a few extra bits to the right of them, and moves the extra bits (which are all zero) in between the important bits such that they appear right after the bits that have to be duplicated.
Phase 6, Duplicating bits: The bits with indices that appear multiple times in L get duplicated. For this a loop is repeated $\log_2(\frac{w}{\log_2 w})$ times. In the first iteration of the loop, all bits whose index appears at least $\frac{w}{\log_2 w} + 1$ times are duplicated. The original stays at its position, the copy is stored $\frac{w}{\log_2 w}$ positions to the right of its original. In the next iteration the same is repeated with bits that appear at least $\frac{w}{2 \cdot \log_2 w} + 1$ times. In addition, the duplicates of the previous steps might have to be duplicated again depending on how often their original's index appears in L. To duplicate bits, first a mask is applied to remove all bits that will not be duplicated in the

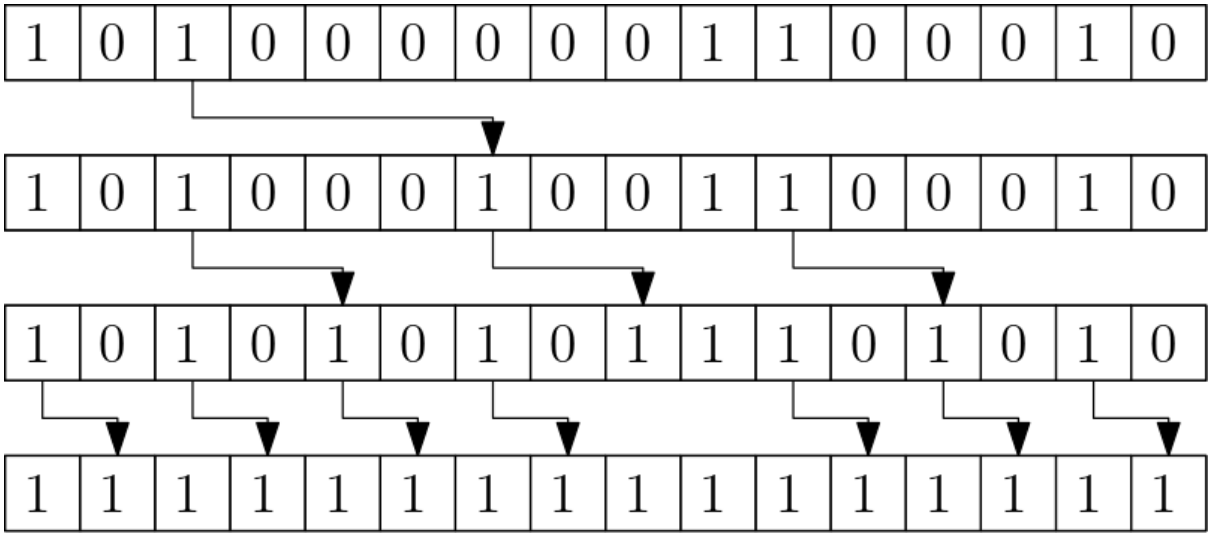


Figure 3.6: Example for step 6, all important bits are 1 at the beginning and every 0 is an empty space created in the previous step.

current step. The remaining bits are shifted to the right and added to the word. The masks for each step are computed during the initialization of the node and require $O(w)$ space. An example for step 6 is depicted in Figure 3.6.

Phase 7, Moving the bits to their final position. All bits at the indices in L are now packed together at the very left and occur as often as they are mentioned in L. However they are not in the correct order, therefore a Benes network is used to move the bits to their final position.

Every step of the bit selector takes at most $O(\log w)$ time and the total number of steps needed is constant, thus in its entirety the bit selector also takes $O(\log w)$ time. An example for the result of each step is shown in Figure 3.7.

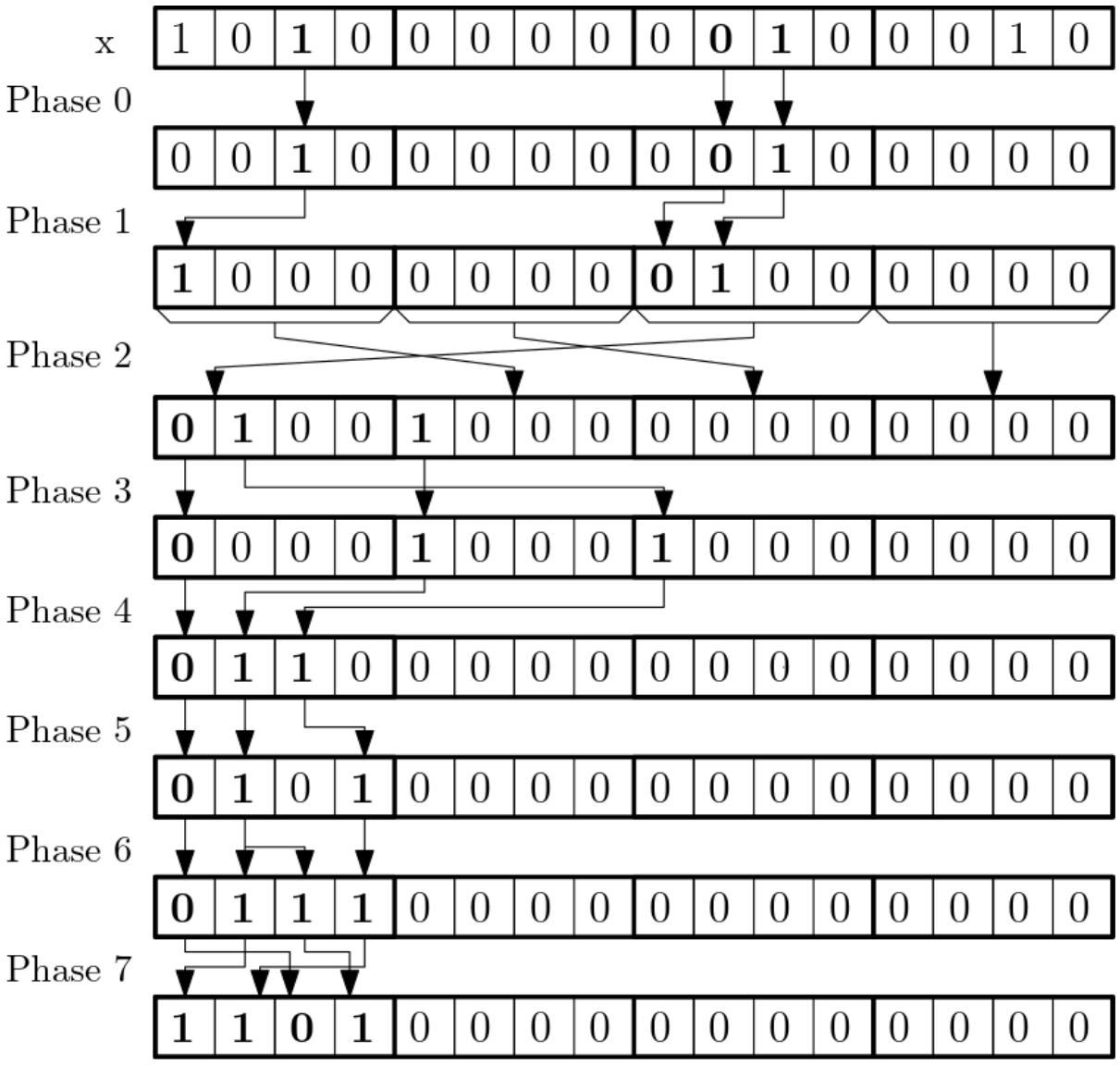


Figure 3.7: Example for all phases of the bit selector with $w = 16$, $x = 1010000000100010_2$ and $L = [13, 5, 6, 13]$.

3.4 Belazzougui and Navarro's Approach

In attachment of the paper "Optimal lower and upper bounds for representing sequences" Belazzougui and Navarro introduce a data structure which supports predecessor queries in time $O(\log \frac{w-\log n}{\log w})$ while using $O(n \log(\mathcal{U}/n))$ memory. Their approach splits the predecessor search into three stages. For the first stage a rank select index is built on a bit array B of size $n + 2^{\lfloor \log_2 n \rfloor}$. The array contains $2^{\lfloor \log_2 n \rfloor}$ bits that are set to zero and n bits that are set to one. The i'th zero is followed by a number of ones equivalent to the number of elements in the data set S that have i as their most significant bits.

In preparation for the second stage, S is split into up to $\lfloor \log_2 n \rfloor$ sets, where each set S_i contains all elements of S that share i as their most significant bits. The elements have their $\lfloor \log_2 n \rfloor$ most significant bits removed before they are inserted into their respective S_i as these bits equal i and are redundant. In the second stage a variation of the van Embde Boas tree (vEB) [4] by Pătraşcu and Thorup [13] is built for each set. Each step in the vEB reduces the key length by half and, in most cases, reduces the search range of the predecessor query significantly. Once all elements of a node fit into one word, the third stage begins. Here the remaining elements, which have been reduced in size because of the vEB tree, are stored consecutively in sorted order and separated by one bit which is set to 1. To find the predecessor of e in the last step, first e is reduced to the bits that are significant at the current node of the vEB tree. These bits are then repeated, with one bit set to 0 in between each repetition, to fill out one word. The predecessor of x can be found by subtracting the word just created using x from the previously mentioned word that has the bits set to 1 in between. After the subtraction, some of the bits which were set to one flip to zero. More precisely, all the bits set to one that followed a element which was smaller than e flip to zero. Because the elements are sorted one can find the first bit which was not set to affected by the subtraction and knows that is is the successor of e. The predecessor is either the successor or next to the successor.

All vEB trees are stored in one consecutive array A sorted by smallest element each tree contains. For every vEB tree, A allocates memory equal to c multiplied by the number of elements the tree contains. c is the minimum value for which all vEB trees fit into their allocated memory, vEB trees that would require less memory are padded. To find the correct vEB tree for a query element q, let q' be the $\lfloor \log_2 |S| \rfloor$ most significant bits of q. The vEB tree containing the predecessor of q starts at index $c \cdot (\texttt{select}_0(B, q') - q')$ in A.

3.5 PGM-Index

The Piecewise Geometric Model index (PGM-index) [5] is a data structure that uses machine learning to create a mapping between the data and its location in the memory. It allows a variety of operations, including fast lookup, predecessor queries and range searches, while using succinct space and also allowing dynamic insert and delete operations. The PGM-index is a fully-learned index that makes use of compression and can be adaptive to both the distribution of the data and the distribution of queries.

4 γ-Node Implementation

Sarel Cohen et al. [3] suggests using the γ-Node on blocks at the end of a predecessor query and another predecessor data structure to find the correct block. In this paper we explore the reversed approach, using a tree of γ-Nodes (γ-Tree) as the structure to find the correct block. To find the predecessor inside a block we us a binary search, but for small blocks linear probing is a valid alternative. This approach was chosen due to the high cache efficiency of accessing elements inside the same block. It doesn't yield any benefit to use a γ-Node instead of a binary search to reduce the required amount of probes on the elements if all probes of the binary search were on elements very close to each other and thus already loaded into the cache (Table 4.1). However, this is not the case for the initial probes a binary search does, where each probe would likely result in a cache miss. It is also important to note that γ-Nodes have a much larger constant omitted by the big-O-notation. In practice, this makes them much slower than binary search on small sets.

To fully leverage the advantages of γ-Nodes, w has to be large (Table 4.2). Currently 32-bit are 64-bit architectures are most commonly used, making γ-Nodes less competitive. Larger value of w can still be used within these systems to increase the capacity of a γ-Node, however doing so leads to longer execution times for fundamental operations such as bit shifts and additions.

4.1 Bit Selector

One of the reasons for the slow performance of γ-Nodes is the high constant factor of the bit selector omitted by the big-O-notation. For $w = 128$, the bit selector has to execute three Benes networks in addition to its own calculations in order to select a maximum of 16 bits from a 128 bit word. Instead, one could use a much simpler bit selector, which we will call a *lazy selector* for its simplicity. The lazy selector picks each bit in L individually and concatenates them. This is achieved in $O(|L|)$ time using a loop. While the asymptotic time is worse, the constant

Index	Search Time
Binary Search	36 ns
γ-Node	536 ns
γ-Node with skip selector	352 ns
γ-Node with lazy selector	244 ns

Table 4.1: Average query time for a γ-Node using different selectors in comparison to binary search using $w = 64$ and $|S| = 8$.

w	$\gamma_{max}(w)$
16	4
32	4
64	8
128	16
256	32
1024	64
2048	128
4096	256

Table 4.2: Maximum number of elements inside a γ-Node ($\gamma_{max}(w)$) for a given word size w.

factor is smaller compared to the selector by Sarel Cohen et al. Because $|L| \leq \frac{w}{\log_2 w}$ the lazy selector requires only $\frac{w}{\log_2 w} \cdot \log_2(w) = w$ bits and is therefore smaller in size as well.

We also implement a third bit selector, which makes use of both the lazy selector and the selector by Sarel et al. It replaces the steps 0 to 5 with a lazy selector, skipping the execution of two Benes networks as well as a few other steps, and will be labelled *skip selector*. Skipping steps 0 to 4 is very simple and the lazy selector does not have to be modified at all. However the fifth step adds spacing between the selected bits for which we have to slightly modify the lazy selector. In addition to L, a second list $\tilde{L}$ which stores how many bits should be empty between $L[i]$ and $L[i+1]$ is stored. The skip selector picks $L[i]$ in ascending order of i, adds it to the temporary result and then shifts the temporary result by $\tilde{L}[i] + 1$. Because the required empty space between two selected bits is never larger than $\frac{w}{\log_2 w}$, the required space to store $\tilde{L}$ is $\log_2(\frac{w}{\log_2 w}) \cdot \frac{w}{\log_2 w}$, which is less than w bits. This provides multiple benefits, first are the reduced space requirements and time constants because the steps 0 to 5 are skipped. The second is the skip selectors asymptotic time, which is no longer in $O(|L|)$ like the lazy selector, but in $O(max(\log w, |\text{unique elements in } L|)$, the first factor $\log w$ is due to step 6 and 7. The second factor comes from skipping step 0 to 5, as the modified lazy selector selects any bits with an index mentioned in L once and moves it to a different, predefined, position. The final benefit only applies if w is larger than the word size of the architecture used but $\frac{w}{\log_2 w}$ is not. In this case step 6 and 7 can still be computed using $\frac{w}{\log_2 w}$ sized words and are therefore relatively fast, while the lazy selector which skips steps 0 to 5 does not slow down significantly if w is larger than the architecture.

4.2 Modified Blind Search

In this thesis the blind search of the γ-Node was modified in multiple aspects. First, the search range reduction was changed because of an assumed oversight in the original paper (subsection 4.2.1). Secondly we reduced the size of Z (subsection 4.2.2). Lastly we added the nodes in the left and right most path of the trie, which are needed for the first iteration of the blind search, to the bit selector. Some optimizations where also added for word sizes larger than the operating system's architecture (subsection 4.2.4).

4.2.1 Search Range Reduction

In the paper by Sarel Cohen et al. [3], each step of the blind search reduces the search range from $[L\ ,\ R]$ to $[L,\ \lfloor\frac{L+R}{2}\rfloor - 1]$ or $[\lfloor\frac{L+R}{2}\rfloor + 1,\ R]$. They also claim that in every iteration both $x[I_L]$ and $x[I_R]$ are known. While one of these two variables can be copied from the last iteration, because either L or R stay unchanged, there is no indication on how $x[\lfloor\frac{L+R}{2}\rfloor + 1]$ or $x[\lfloor\frac{L+R}{2}\rfloor - 1]$ could be found. However, the subtraction/addition of 1 could be removed and $\lfloor\frac{L+R}{2}\rfloor$ as the left/right bound used for the next iterations search range instead. In this case $x[\lfloor\frac{L+R}{2}\rfloor]$ can be easily found because it is calculated in the current iteration. Asymptotically the search range still converges logarithmically but it requires more loop iterations. The termination condition also has to factor in the size of the search range because for the range $[L,\ L+1]$ q equals L and the range is unable to converge to $[L+1,\ L+1]$.

4.2.2 Smaller Z

To calculate $y_q[I_q]$ a section of either $y_q[I_{L_q}]$ or $y_q[I_{R_q}]$ is concatenated with $y_q[\tilde{I}_q]$ according to Sarel Cohen et al. [3]. This does not work. While q and R or L share a path up to an inner node u, they only share their prefix up to the index right before the index of u. This is because u is the node at which the two paths separate and thus the bit has to be different. As a result there is one bit missing when computing $y_q[I_q]$. This bit can be easily computed, because it is known whether the first part of $y_q[I_q]$ is copied from $y_q[L]$ or $y_q[R]$. If $y_q[L]$ was picked, the bit has to be 1, because L is to the left of q, and thus the path to q has to split to the right side. For the same reason the bit is 0 if R was chosen.

4.2.3 Smaller Benes Networks

In the paper by Sarel et al. [3] the Benes networks are realized by using two bit arrays for every layer. The `direct` bit mask is used to extract all bits that do not change position in the layer and the `crossed` bit mask for all bits that do change positions. Storing both is unnecessary because `crossed` and `direct` are the same except every bit is flipped. Another trick, which was not implemented in this thesis, halves the necessary memory again. The first and second half of the bit arrays are the same. Thus one can only store the first half and reconstruct the entire bit array by shifting the first half and then adding it to itself. When one concatenates the bit arrays of every layer together, after using both methods to reduce memory, only $\frac{w}{2}$ bits are required, which is equal to the total number of switches in the Benes network and thus optimal. Because our implementation used different values for w, there was no convenient way to store the $\frac{w}{2}$ bits in a data type of the same size. Thus we choose not implement the second size reduction and use the data type for w that was already available.

4.2.4 Optimizations for large word sizes

One of the word sizes we intensively used for the γ-Node is 1024 because $\frac{1024}{\log_2 1024} = 64$, which is the word size of the architecture we used in our testing environment. The two bit strings $y_q[I_q]$ and $x[I_q]$ used in each step of a blind search iteration are the dependent on I_q, which is the path from the root of the trie to q and is always smaller than $|S|$. Therefore, generating

$y_q[I_q]$ and $x[I_q]$ does not require emulating larger word sizes. The trie is also small enough that the size of the concatenation (Z) of all $\tilde{I}_q$ is less or equal 64 bit and thus we can store Z within one 64 bit word. The only sections where arithmetic operations on 1024 bit words is needed are the generation of z, because z is a 1024 word, and the phases 0 to 4 of the bit selector. While some of the data for the bit selectors is stored in 1024 bit words, it can be extracted without using operations on 1024 bit words by first addressing the bytes that contain the section of the data one needs, which will be spread over less than 8 byte, and only shifting the 64 bit word that contains the entire section.

There is also the problem of accessing individual bits. For small values of w, a bit shift can be used to access a specific bit. However, this can be a significant time loss if a machine's architecture is smaller than the word size. Let w' be the word size used by the machine's architecture. An element can be split into $\frac{w}{w'}$ blocks of size w'. One can calculate the block that contains the bit easily. An individual bit can be accessed by first extracting the block that contains it by using pointer arithmetic. Then a bit shift on the block can be used to access the bit itself.

4.3 γ-Tree

γ-Nodes on their own do not have many applications due to the limited size of S. To remove this limitation we introduce the γ-Tree, which is a tree with a high branching factor, elements of S in its leaves and γ-Nodes as its inner nodes. Let $\gamma_{max}(w)$ be the maximum capacity of a γ-Node using word size w. To build a γ-Tree, split S into blocks of size $\gamma_{max}(w)$ and build a γ-Node on every block. Next build a γ-Nodes on every $\gamma_{max}(w)$ consecutive blocks, using the smallest element of each block as representative for the block. This process is repeated until a single γ-Node, called root, is responsible for all elements. A predecessor search starts at the root and can identify the child of the root that holds the predecessor in $O(\log w)$ time while probing S a maximum of three times. The search continues in said child of the root and repeats this until it reaches a leaf, which will either contain the predecessor or the successor. With the γ-Tree having a height of $\log_{\gamma_{max}(w)} |S|$, this requires $O(\log_{\gamma_{max}(w)} |S|)$ probes on S. The time a predecessor query takes is

$$\begin{aligned} O(\log w \cdot \log_{\gamma_{max}(w)} |S|) &= O(\log w \cdot \log_{w/\log w} |S|) \\ &= O(\log w \cdot \frac{\log |S|}{\log(w/\log w)}) \\ &= O(\log w \cdot \frac{\log |S|}{\log w - \log^2 w}) \\ &= O(\frac{\log |S|}{1 - \frac{\log^2 w}{\log w}}) \end{aligned} \tag{4.1}$$

and can be reduced to $O(\log |S|)$ for $w > 1 + \epsilon$.

4.3.1 Cutoff

While the just described γ-Tree minimizes the number of probes and has the same time complexity as binary search, practically the γ-Nodes on the lower levels of the tree are not as efficient as those near the root. This stems from the fact that these nodes are responsible for elements that are stored closer to each other. If they are close enough, probes on them cost very little time making algorithms such as binary search and linear probing much faster, meanwhile the γ-Nodes large overhead stays the same. Let us call the block size at which a γ-Tree switches from using γ-Nodes to binary search or linear probing the cutoff c. Because of the high branching factor of a γ-Node, by far the most allocated space of a γ-Tree is used by its lowest level. As a result the cutoff is highly relevant when calculating the overall space allocated by a γ-Tree. On average, doubling the cutoff will halve the redundant space.

There are two ways to implement the cutoff. First is a bottom up approach, the γ-Tree splits the input into $\lceil \frac{|S|}{c} \rceil$ blocks and builds γ-Nodes on top of these blocks. In this case the cutoff is static and does not depend on the input size. This approach often leads to the γ-Nodes at the root not reaching its full capacity. It causes the time for predecessor queries to change little when the input size is increased, up until a point where the current height of the γ-Tree can no longer support it and another layer is added, creating sudden jumps in its query times.

The alternative is a top down approach, in which first the root is created with the biggest capacity possible, then on then the children of the root and so on, until a child would have less than c leaves bellow it. At this point the cutoff is reached and the tree will switch to binary or linear search. Because γ-Nodes have a high branching factor the cutoff can be overshot by a lot. Using this approach leads to smoother times for predecessor queried, however it also introduces jumps in the allocated memory because after a cutoff is reduced, a lot of elements can be inserted without adding another layer to the γ-Tree by increasing the cutoff.

In theory the top down approach should perform better in terms of speed. The reason for this is simple as a top down approach guarantees that as many nodes as possible are at high capacity, which is optimal because a γ-Nodes performance is practically independent of its capacity. Meanwhile, the root in the bottom up approach is often at less than half capacity and thus a tree of the same high could lead to a smaller search range for the binary search, but doesn't because the root's branching factor is lower than it could be. For example, using $w = 128$ and thus a maximum branching factor of 16, $c = 32$ and $n = 100$ the bottom up approach creates 3 blocks of size 32 and one block of size 4, resulting in a branching factor of 4 in the root. The top down approach on the other hand would create 12 blocks of size 8 and one block of size 4, leading to a branching factor of 13. Thus more often than not the predecessor query, using one γ-Node, reduces the search range to only 32 in the first case and 8 in the second.

However, in practice the cache has to be considered as well. While a γ-Tree with nodes at low capacity does not reduce the search range as much as it could if its node were at high capacity, the lowest level of the tree consist of less nodes. Fewer nodes mean that the tree might fit into the cache in its entirety and thus will cause no cache misses, while a bad allocation strategy might cause a lot of cache misses if the tree is too big. Should n be big enough the specific cutoff, and whether it is dynamic or static, becomes less relevant. They only add a constant overhead, which is overshadowed by the operations needed to traverse the rest of the tree.

Our tests showed that the value for the cutoff is very lax when it comes to its impact on the time a predecessor query takes (Figure 7.2). In general, it is beneficial to use a high cutoff because the memory requirements are lower.

4.3.2 Node structure

The nodes of a γ-Tree are stored inside an array using the same structure has a heap. All nodes on the same level of the tree are stored consecutively by the same order as the elements they contain. The root is at the start of the array, followed by all nodes that are children of the root, followed by all the children of these nodes and so on. Let q be the node at which the predecessor query is currently in the tree, j the size of the blocks in q, s the index of the first node on the level below q and $r \in [0, |S|)$ be the rank that q returns as predecessor. The node in the next level that the predecessor query will look at is at index $\tilde{i} = s + \frac{r}{j \cdot \gamma_{max}(w)}$. Using $\tilde{i}$ the γ-Tree can be stored without any pointers to γ-Nodes.

5 Analyzing Belazzougui and Navarros Approach

The predecessor search described in the attachment of "Optimal lower and upper bounds for representing sequences" [2] is split into three parts. In the first part the word size is reduced from w to $w - \lfloor \log_2(n) \rfloor$. Elements are split into sets using the first $\lfloor \log_2(n) \rfloor$ bits as separator. A rank select data structure is used to find the set which holds the predecessor. In the attachment a variation of the vEB tree [4] by Pătraşcu and Thorup [13] is used to continue the predecessor search on the sets. They use a vEB tree because of its fast asymptotic time, however, many other predecessor queries could be used as well. The sets on which the vEB trees are built can be as small as one element and the sizes and number of sets depends on the distribution of elements.

Let the distribution be uniform random and the number of elements n a power of two. Let S' be the set of unique prefixes of length $\log_2(n)$ of the elements in S. The proportion of elements that are in S' in relation to all possible elements with $\log_2(n)$ bits is:

$$\frac{|S'|}{2^{\log_2 n}} = \frac{2^{\log_2 n} \cdot (1 - (\frac{n-1}{n})^n}{2^{\log_2 n}} = 1 - (\frac{n-1}{n})^n \overset{n \to \inf}{=} \frac{e-1}{e} \approx 0.632 \tag{5.1}$$

and therefore the average vEB tree contains 1.58 elements. In this case it is much more practical to use a linear or binary search because the number of elements is so low.

An opposite distribution would be one in which every element has the same first $\lfloor \log_2(n) \rfloor$ bits. In this case all elements end up in the same set and the rank select data structure's only benefit is the reduction of the word sizes within the aforementioned set. Using a vEB tree can be beneficial in this case as its speed is more dependent on w than on the size of the set itself.

It is important to note that Belazzougui and Navarro's approach is best used for small universes and a large n as its size of $O(n \cdot (\log(u) - \lfloor \log n \rfloor)$ and its speed of $O(\log \frac{\log(u/n)}{\log^2 u})$ benefit from both.

The memory needed for a vEB tree of a single set is $c(|S_i|(\log u - \log n'))$ where c is some constant. The bit array for the rank select data structure requires $2^{\lfloor \log_2(n) \rfloor} + n$ bits. The rank select data structure itself has to be stored as well, which can be achieved with less 5% of the memory the bit array requires [19].

The size of the overall data structure is highly dependent on c, which in turn dependents on w and n, for example using $u = 2^{128}$ and $n = 2^{32}$, the sizes of all vEB trees combined is:

$$\begin{aligned}
&\sum_i c(|S_i|(\log u - \log n')) \\
&= \sum_i |S_i| \cdot c(\log u - \log n') \\
&= n \cdot c(\log u - \log n') \\
&= 2^{32} \cdot c(128 - 32) \\
&= 2^{32} \cdot c \cdot 96
\end{aligned}$$

As a result, for $c = \frac{4}{3}$ the memory usage by the vEB tree is the same as the memory used for storing the elements of S. This is fine as the vEB trees store the elements inside them as well. However, for $c = \frac{8}{3}$ the allocated redundant memory would be as big as the memory required to store the elements of the set.

Each node in the vEB tree, responsible for elements with w' bits, consists of three parts, a summary node, a collection of children and a dictionary to find its children. This dictionary uses the first half of bits of an element as key and stores five values, these are the minimum and maximum element in the child as well as their ranks and a pointer to the child. The minimum and maximum element need w' bits, their rank $\log_2 n$ bits each and finally the pointer to the child. This pointer is stored differently and only uses one bit in the dictionary, indicating if it is a null pointer or not.

For example, for $S = [0 \cdot 2^{78}, 1 \cdot 2^{78}, ..., 2^{25} \cdot 2^{78}]$ and $w = 128$, S is the set that contains all elements with the first 25 bits and last 78 bits set to zero. For this set the predecessor structure of Belazzougui and Navarro [2] will store all elements in a single vEB tree. The first node of this tree would have a 2^{25} children. For every child the dictionary stores an entry, which has size $3 \cdot 51 + 2 \cdot 25 + 1 = 204$. In addition a summary node is build on the upper 48 bits. The summary will again have 2^{25} children and entries in the dictionary require $3 \cdot 25 + 2 \cdot 25 + 1 = 126$ Bits. Therefore at least 329 bits are required for every element in S, which is an overhead of over 150%.

While it is unlikely this exact scenario occurs, finding distributions that often cause c to be quite large is not hard. If 2^w is sufficiently larger than n, any vEB tree with 3 elements that differ in their most significant bits has a large overhead. This is due to both the root and its summary having 3 dictionary entries. The rank select data structure expects c to be the same for all sets S_i. If a single vEB tree requires requires c to be large, all other vEB trees have to padded so that their size is c multiplied with the number of elements they contain. This is due to the method the rank select data structure uses to finds the set that contains the predecessor. It calculates how many elements are smaller than the smallest element in the vEB tree it wants to access and then multiplies this number with c and $w - \lfloor \log_2(n) \rfloor$, the result is the position at which the vEB tree with the predecessor will start.

For some distributions, vEB trees can also have a relatively low space requirement, even lower than the original data. For example, when all elements in S share their first $\frac{w}{2}$ most significant bits, the root only has a single child. All elements in this child are of size $\frac{w}{2}$, effectively reducing the size of all elements in S by half using a constant number of bits.

Because of the interaction with c, the great performance of rank select data structures on their own (Figure 7.3) and time constrains, we did not implement the rank select data structure paired with the vEB tree as described by Belazzougui and Navarro. Instead, we implement two predecessor data structures, the rank select data structure paired with binary search (section 6.2) and the vEB tree on its own. Due to time constrains, the dictionaries in the vEB tree were stored separately from the nodes, with every node storing a pointer to its dictionary. Instead of a tuple it also only contains a pointer to the position of a child, and said child contains the information normally stored in the tuple. This adds a constant overhead because more pointer are stored. However this also reduced memory. When the algorithm switches to parallel comparison, it is no longer needed to store both the minimum and maximum rank because the distance of the two can be extracted from the word used for the parallel comparison. On the other hand, a tuple would have to store both values anyways because, as a member of a dictionary, it has to be of fixed size.

6 Other Implementations

6.1 Fusion Trees

The word sizes used in this paper do not allow a large branching factor for fusion nodes. Using non-perfect sketches would reduces the branching factor even further, to a point were it becomes redundant to use fusion trees in the first place. For this reason we are using perfect sketches, for which we use the lazy selector introduced in section 4.1. By using perfect sketches the fusion nodes achieve a branching factor 7, 10 or 31 for $w = 64, 128$ or 1024 respectively. In addition we use the same structure as the γ-Tree and switch to a binary search once the search range is small enough.

6.2 Rank Select

A rank select data structure on the first m bits can drastically reduce the search range. Using the bit array a decribed by Belazzougui and Navarro [2] (section 3.4), the search range of a predecessor query for x can be reduced to the range:

$$\texttt{start} = \texttt{select}_0(a, x >> (w - m)) - (x >> (w - m)) + 1 \tag{6.1}$$

$$\texttt{end} = \texttt{select}_0(a, (x >> (w - m)) + 1) - (x >> (w - m)) \tag{6.2}$$

were $>>$ is a bit shift to the right. One can now use a binary search on the range [start, end] or begin a linear search at start. A linear search can be faster if the distribution is sufficiently uniform because end is not needed and does not have to be computed. However, it is much slower for distributions concentrated around a few points. For this reason we will be only using binary search in this thesis.

7 Results

The predecessor data structures are evaluated on their space efficiency and their speed. In this thesis we evaluate the data structures for different word sizes, data distributions and limitations on the working memory.

7.1 Setup

The machine used for our experiments has two Intel Xeon ES-2650 v2 with a total of 12 cores that run at 2.3 GHz. It uses a 30 MB Cache and a SATA 10000 hard drive as well as 128 GIB DDR4-RAM. All implementations are written in C++ and single threaded.

In each experiment we measure the time it takes to search the predecessor of 10000 elements and divide this number by 10000 to get the average predecessor search time. This was repeated 50 times and the median used as data point. The query elements and the elements in the predecessor data structure are drawn from the same distribution, with 1% of the query elements also appearing in the set for the data structure.

We evaluate the predecessor data structures for three different word sizes. The first and smallest word size is 64 bit. It was chosen because the architecture of the system for our experiments is also 64 bit, allowing fast elementary operations like bit shifts and additions while still giving us a branching factor of 8 for γ-Nodes and 7 for fusion nodes. The second word size is 128 bit, because there is a native integer data type using 128 bit in C++, still enabling fast operations but increasing the branching factor. Lastly we use a word size of 1024 bit as it facilitates some optimizations for the γ-Nodes (see subsection 4.2.4) and increased the branching factor to 64 for the γ-Node and 31 for the fusion node.

The data set S used for the experiments is generated from three different distributions. The first follows a uniform random distribution generated using the standard c++ library. The second data set uses the longitude coordinates of points of interests in Italy and is taken from OpenStreetMap [12]. This data set is only used for word size 64 because of its limited precision. Lastly we use a set of URLs from 2001 obtained by the WebBase crawler [15] and filtered by the Laboratory for Web Algorithmics [9]. All URLs in this data set start with the prefix "http://". Because at least one predecessor data structure is heavily influenced by the distribution of the most significant bits we decided to split it into two. One with the original data and one with the prefix "http://" removed. In addition, the URLs are of different length. URLs that can be stored using less than 1024 bits have their remaining bits padded with 0. URLs that require more than 1024 bit have their suffix cut of. If the removal of a suffix creates a duplicate, it is removed from the set.

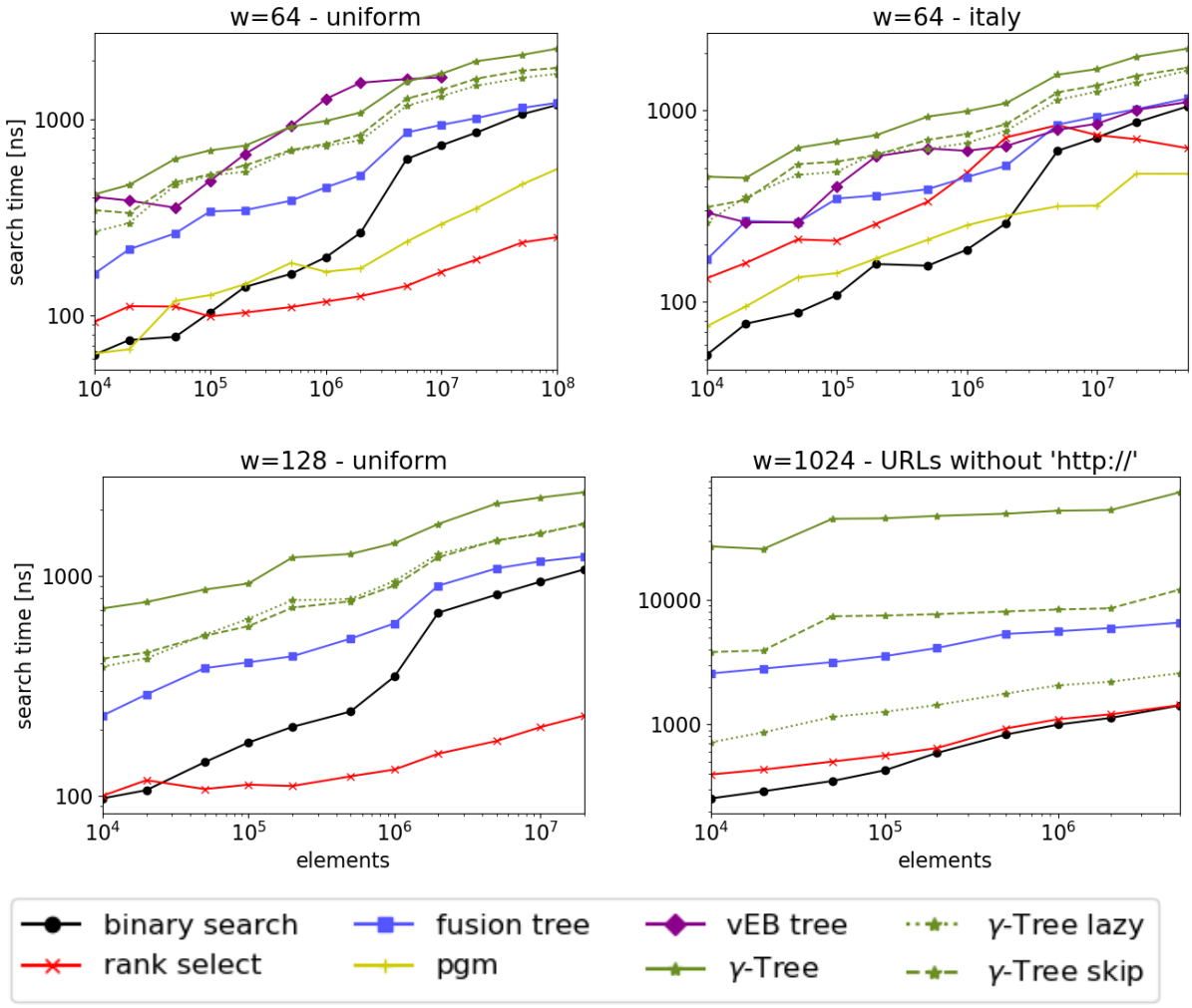


Figure 7.1: Speed of the predecessor queries on external memory

7.2 Implementations

For the rank select predecessor data structure we use the implementation by Florian Kurpicz [8] for operations on the bit array. An implementation of the PGM-index was made public by its authors. It can be found on `github.com` [14] and is used in our experiments. For the data type with a word size of 1024 bit a generic C++ template from the Wide-Integer repository [16] by Christopher Kormanyos is used.

7.3 Experiments

Experiments on internal memory In our first experiments (Figure 7.3) we make full use of the working memory of the machine. Of the three selectors for γ-Nodes, the lazy selector performs the best in terms of speed on a 64 bit architecture. However, it is clear that its performance lags far behind existing predecessor queries like the PGM-index or binary search. The rank select data structure that switches to binary search has one of the fastest performances, and is

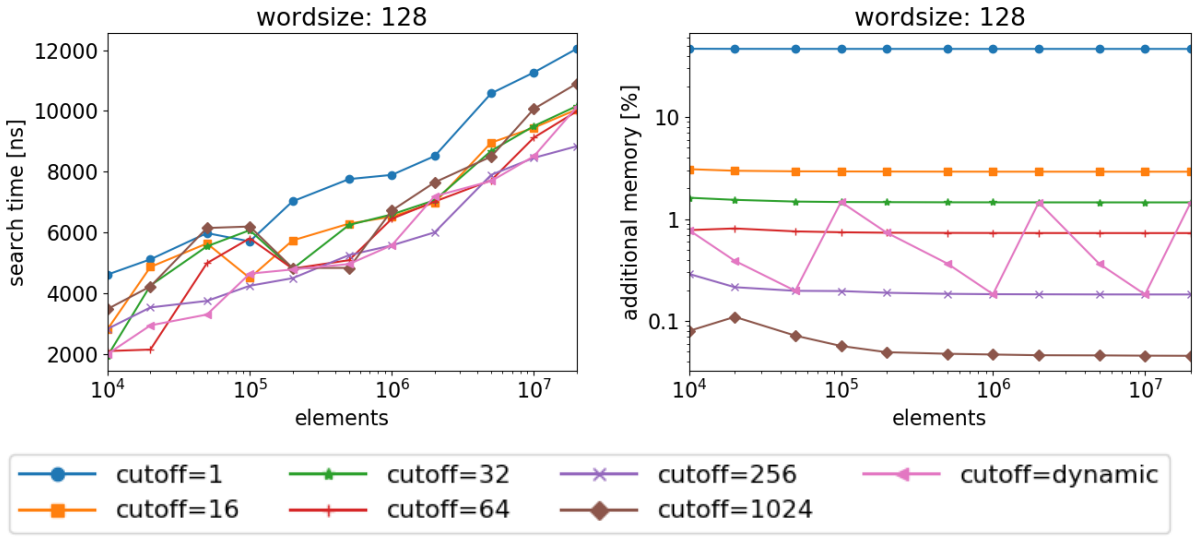


Figure 7.2: Performance of the γ-Tree with a lazy selector for different static cutoffs as well as a dynamic cutoff of at least 32 on a uniform distribution with w=128.

faster than the PGM-index for a uniform distribution. However it is much more dependent on the distribution of the data compared to a γ-Tree and fusion tree.

Experiments on external Memory. To fully leverage the low number of element probes of the γ-Tree, we artificially reduce the working memory the machine has access to. We store data for the predecessor index as well as S in two files on the hard drive. This severely slows down the time it takes to access data. We also add a small cache of 4096 Bytes to both of these files which will store the last accessed element as well as its neighbours. This allows faster access times if consecutive data probes are in one vicinity. Unfortunately, the public implementations for the PGM-index, the modified vEB tree and the rank select data structure do not support such a change. For this reason, the PGM-index and the modified vEB tree are excluded from the remaining experiments. The rank select data structure on the other hand will be included because the binary search in the second part is still affected by our change, however it should be noted that its first part still operates with no limitations and has an unfair advantage because of it.

Our first experiments (Figure 7.2) using external memory are to determine the cutoff used for the γ-Tree. As one can see in the graph, the cutoff is very lax and the performances relatively even between 32 and 256. Because of this we decided to use a static cutoff of 256 as it has the least amount of redundant data to store. It is also interesting to note that the search time does not always increase when the number of elements increases. This is a side effect of the external memory and does not occur our tests for internal memory.

In our next experiment using external memory we compare the predecessor search time of the data structures. Except for the very skewed distribution of the URLs with "http://" as prefix, the rank select data data structure performs the best. Nevertheless it has be noted again that we were unable to limit the working memory the rank select structure itself uses because we used an existing implementation, making it much faster than it would normally be. For $w = 64$ and $w = 128$ the γ-Trees with different bit selectors are close to each other as they

all need the same amount of probes on the elements. For $w = 1024$ this changes because the memory requirements for the nodes differs much more and, for the bit selector of Sarel et al., computations are very slow because of the large word size.

The fusion tree performs similar to binary search because of its low branching factor. It has to probe more elements in S and on the tree. In addition, a fusion node often requires one more probe on the elements than its γ-Node counterpart would.

Memory Requirements. Lastly we evaluate the memory required to store the different data structures. The memory required to store the indices is divided by memory used to store S. S itself is not compressed and each value in S is stored individually, thus the total number of bits used to store S is $|S| \cdot w$. Our experiments show that the PGM-index is by far the most space efficient algorithm for $w = 64$.

For $w = 64$ and $w = 128$ the fusion tree is slightly more space efficient than the γ-Trees. While there are more nodes in the fusion tree because of a lower branching factor, the nodes themselves requiring less space. At a word size of 1024 bit the branching factor of a fusion node is less than half of that of a γ-Node resulting in the γ-Tree with a lazy selector using less memory than a fusion tree. For $w = 64$ and $w = 128$ the γ-Trees using the original bit selector by Sarel et al. and the ones that use the skip selector require the same amount of space. In theory the skip selector would allocate less memory for these word sizes, but does not because of byte padding by the compiler.

There are two plot lines for the vEB tree described in "Optimal lower and upper bounds for representing sequences" [2]. One is for the memory used by the implementation of this paper. The other plot line shows the theoretical memory usage if no pointers to the dictionaries are stored and the values of the dictionaries scaled with the depth of the tree. These optimizations were scratched due to time constrains. The plots lines show that for a uniform distributions the vEB tree requires a significant amount of memory. This is caused by the large volume of dictionary entries with one element in the root. We had to prematurely end the experiments at $|S| = 10^7$ because our machine could not handle the high memory usage. However, if the distribution is skewed, the vEB tree can require less memory than $|S|$ itself. Because all values of S are stored inside the tree, technically one could deallocate S to reduce the total memory used to below $w \cdot |S|$. However, this comes at the cost of other operations, such as accessing elements at a specific index, which would no longer be possible in constant time.

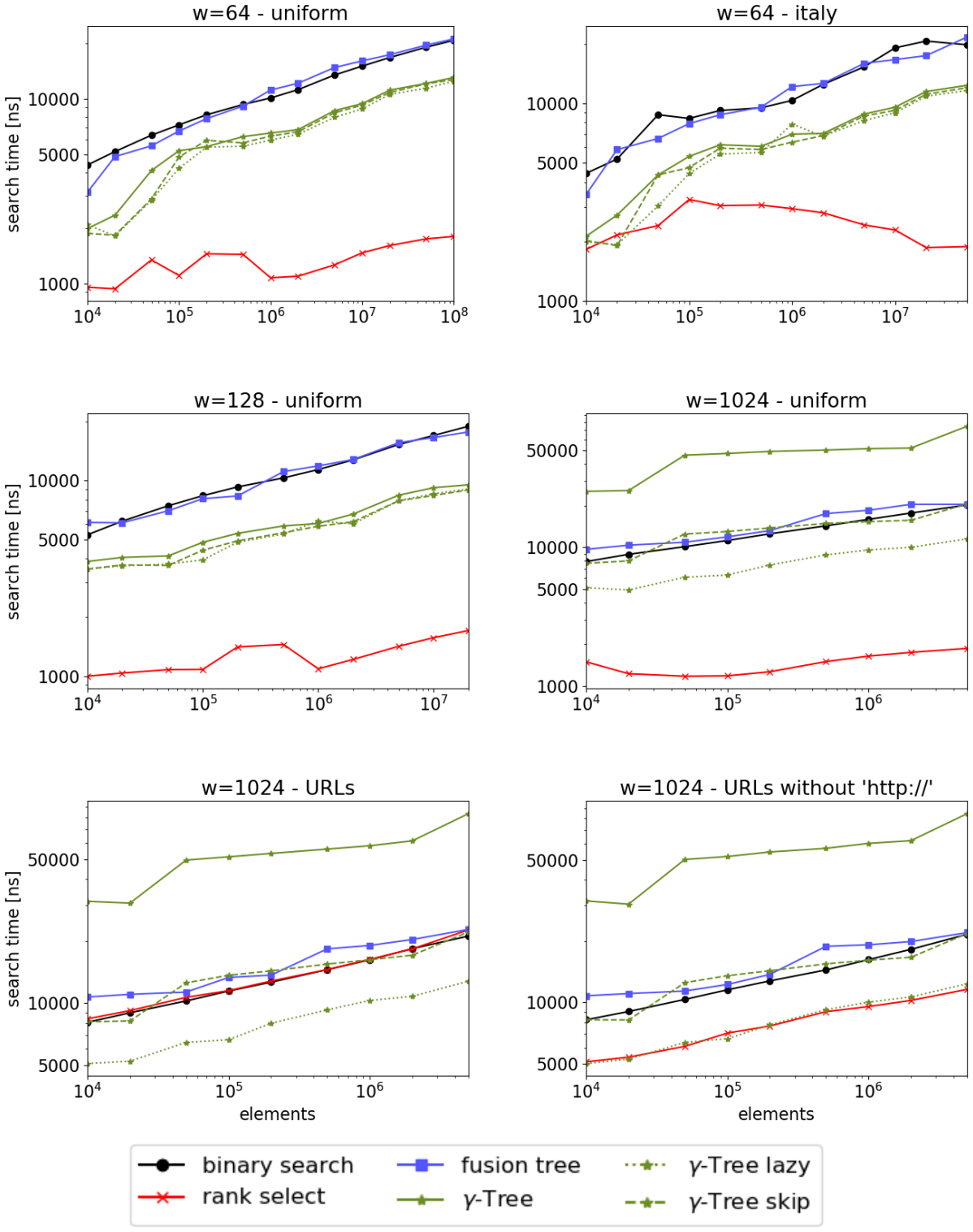


Figure 7.3: Speed of the predecessor queries on external memory.

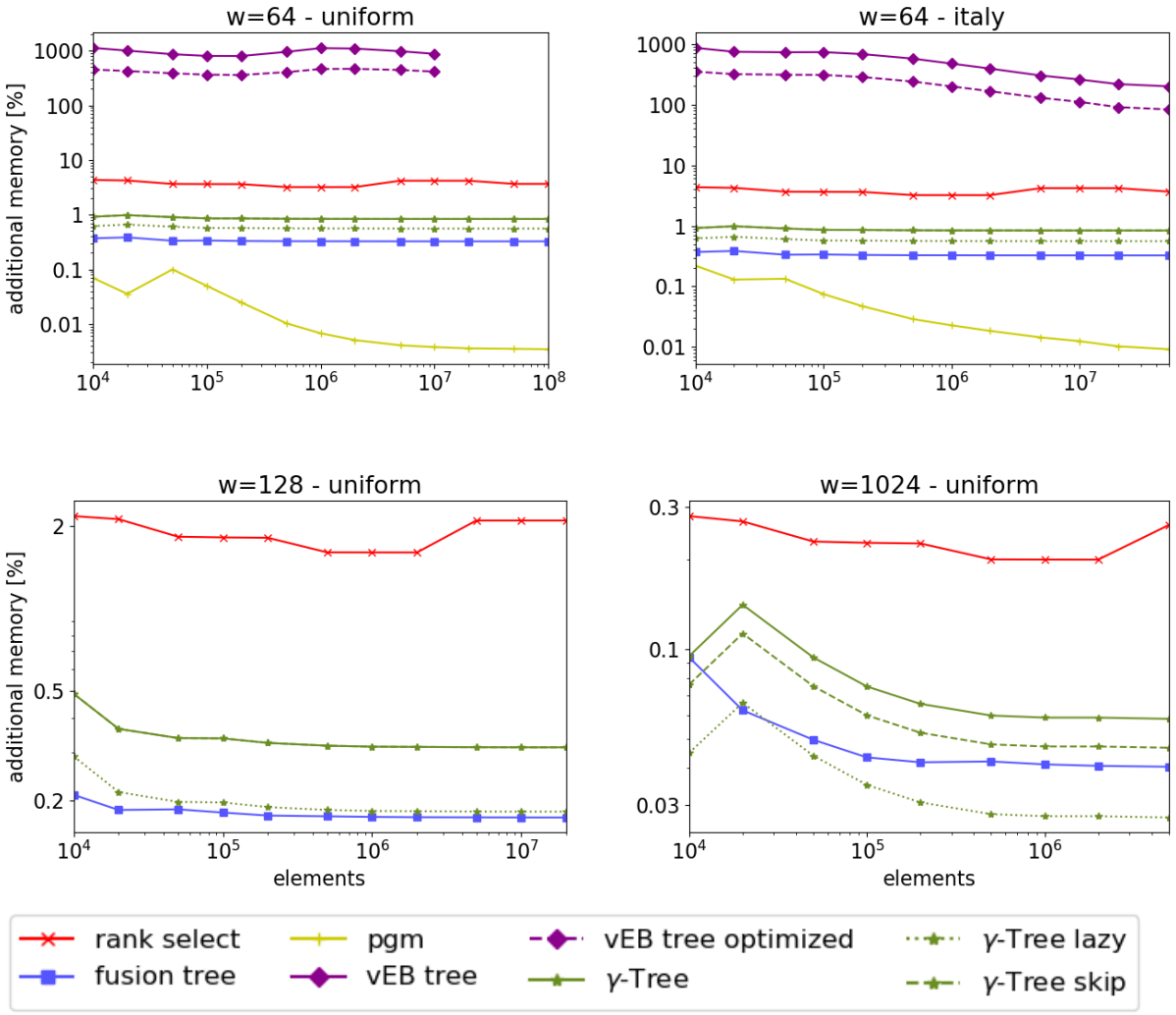


Figure 7.4: Additional memory used by the different predecessor queries in relation to the memory used to store the elements.

8 Conclusion and Future Work

In this work, we implemented the succinct predecessor data structure by Sarel et al. called γ-Node [3]. However due to the, in this context, small bit size of currently available architectures and the high overhead, the γ-Node performed poorly in the average scenario. By storing the data on external memory instead, we showed that γ-Nodes can be useful if memory access is very slow. Yet we were not able to compare them to implementations other than our own in most tests because the predecessor queries that were publicly available were either not compatible with the data structure used for external memory accesses or with the large word sizes. It is likely that the PGM-index will perform better than the implemented γ-Trees in these scenarios as well. Our benchmarks also showed that the bit selector used by Sarel et al. is too slow and a simpler solution performs better in practice.

The succinct predecessor data structure in the attachment of "Optimal lower and upper bounds for representing sequences" [2], while asymptotically succinct, is very memory intensive in practice if the universe is much larger than the data set. However, the rank select structure used in this attachment can be used in combination with an in-place algorithm like binary search. For sufficiently uniform distributions this combination achieves great performances. The combination of a rank select data structure and a vEB tree does not work quite well because the size per element has to be the same for all vEB trees.

Future Work. Probably the most competitive predecessor query implemented in this paper was the rank select data structure. Its main problem is the requirement of a semi uniform distribution on the most significant bits. This problem could be reduced by using a entropy compressed bit array [11] which would allow an increase in the size of the bit array and a better reduction of the search range. A different approach could be to expect the data to follow a distribution that is concentrated around a few points and build a separate rank select structure for each such point. This would allow the removal of the first bits as all elements around such a point would share them.

The rank select data structure had an unfair advantage in the tests with external memory because we used a public implementation which did not support this. Its performance without such an advantage could be a topic of future research.

In this thesis we were limited to using a machine with a 64 bit architecture. While unlikely, should higher bit architectures become more common, it might be interesting to take another look at γ-Nodes, as it is a severe drawback for them.

Bibliography

[1] Paul Beame and Faith E Fich. "Optimal bounds for the predecessor problem and related problems". In: *Journal of Computer and System Sciences* 65.1 (2002), pp. 38–72.

[2] Djamal Belazzougui and Gonzalo Navarro. "Optimal lower and upper bounds for representing sequences". In: *ACM Transactions on Algorithms (TALG)* 11.4 (2015), pp. 1–21.

[3] Sarel Cohen et al. "Minimal indices for predecessor search". In: *Information and Computation* 240 (2015), pp. 12–30.

[4] Peter van Emde Boas. "Preserving order in a forest in less than logarithmic time". In: *16th Annual Symposium on Foundations of Computer Science (sfcs 1975)*. IEEE. 1975, pp. 75–84.

[5] Paolo Ferragina and Giorgio Vinciguerra. "The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds". In: *Proceedings of the VLDB Endowment* 13.8 (2020), pp. 1162–1175.

[6] Michael L Fredman and Dan E Willard. "Surpassing the information theoretic bound with fusion trees". In: *Journal of computer and system sciences* 47.3 (1993), pp. 424–436.

[7] Michael L Fredman and Dan E Willard. "Trans-dichotomous algorithms for minimum spanning trees and shortest paths". In: *Proceedings [1990] 31st Annual Symposium on Foundations of Computer Science*. IEEE. 1990, pp. 719–725.

[8] Florian Kurpicz. "Engineering Compact Data Structures for Rank and Select Queries on Bit Vectors". In: *SPIRE*. Vol. 13617. Lecture Notes in Computer Science. Springer, 2022, pp. 257–272. DOI: 10.1007/978-3-031-20643-6_19.

[9] *Laboratory for Web Algorithmics*. https://law.di.unimi.it/webdata/webbase-2001. Accessed: 2023-6-13.

[10] J Ian Munro et al. "Succinct representations of permutations". In: *Automata, Languages and Programming: 30th International Colloquium, ICALP 2003 Eindhoven, The Netherlands, June 30–July 4, 2003 Proceedings 30*. Springer. 2003, pp. 345–356.

[11] Daisuke Okanohara and Kunihiko Sadakane. "Practical entropy-compressed rank/select dictionary". In: *2007 Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM. 2007, pp. 60–70.

[12] *OpenStreetMap*. https://www.openstreetmap.org/. Accessed: 2023-01-10.

[13] Mihai Pătraşcu and Mikkel Thorup. "Time-space trade-offs for predecessor search". In: *Proceedings of the thirty-eighth annual ACM symposium on Theory of computing*. 2006, pp. 232–240.

[14] *PGM Implementation*. https://github.com/gvinciguerra/PGM-index. Accessed: 2023-3-19.

[15] *The Stanford WebBase Project*. `http://diglib.stanford.edu:8091/~testbed/doc2/WebBase`. Accessed: 2023-6-19.

[16] *WideInteger*. `https://github.com/ckormanyos/wide-integer`. Accessed: 2023-3-19.

[17] Dan E Willard. "Log-logarithmic worst-case range queries are possible in space Θ (N)". In: *Information Processing Letters* 17.2 (1983), pp. 81–84.

[18] Dan E Willard. "Log-logarithmic worst-case range queries are possible in space Θ (N)". In: *Information Processing Letters* 17.2 (1983), pp. 81–84.

[19] Dong Zhou, David G Andersen, and Michael Kaminsky. "Space-efficient, high-performance rank and select structures on uncompressed bit sequences". In: *Experimental Algorithms: 12th International Symposium, SEA 2013, Rome, Italy, June 5-7, 2013. Proceedings 12*. Springer. 2013, pp. 151–163.