An Investigation of Hierarchical Bit Vectors

James Glenn David Binkley
Loyola College in Maryland
{jglenn,binkley}@cs.loyola.edu

ABSTRACT

Hierarchical bit vectors represent sets of integers using a collection of bit vectors. At the top level, a single bit is set iff the set is non-empty. The bits of this next level summarize ranges of the elements. In the case of a binary hierarchical bit vector the two bits of the next level summarize two ranges: the lower half and the upper half of the possible elements. At the lowest level each bit records the membership of a particular integer.

Hierarchical bit vectors find application in information retrieval, bioinformatics, non-averaging sets, and the conversion of *NFA*s to *DFA*s. Competing data structures for such applications include simple bit vectors and tree-based structures such as skip-lists. A comparison of hierarchical bit vectors with two other representations (simple bit vectors and binary search trees) is presented. The comparison includes both analytical and empirical analysis of the hierarchical bit vectors.

The analytical results show that as the size of the set used increases, hierarchical bit vectors enjoy an advantage over tree-based structures and that as the sets used become sparser, hierarchical bit vectors perform better than standard bit vectors. The empirical results confirm that hierarchical bit vectors sit in between the other two. The empirical investigation also highlights the impact that the processor cache has on the break-even points.

1 INTRODUCTION

A *hierarchical bit vector* (*HBV*) is data structure for storing sets of integers in some fixed range $0, \cdots, N-1$. *HBV*s are useful as a compromise between standard bit vectors, which can perform set insertions and deletions in constant time, but are slower to enumerate the elements of a set, and tree related structures where insertions and deletions require logarithmic time, but enumeration is quick.

A *HBV* can be viewed as a sequence of bit vectors arranged in layers. At the lowest layer is a vector of N bits. Each bit at this level records the membership of a particular integer in the set – this is a standard bit vector. The next higher level is a bit vector of N/k bits, each bit of which summarizes a group of k consecutive bits of the level below. A bit at this level is set iff any of the summarized bits are set. Consecutive layers summarize groups

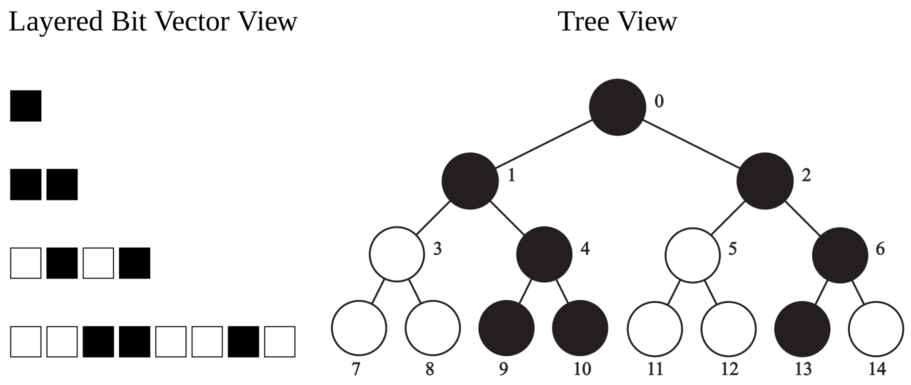

Figure 1: An *HBV* representing $\{2, 3, 6\}$

of k bits from the next lower level until a layer of a single bit is produced. This bit records whether or not the set is empty. The value of k can differ on different levels.

For example, consider the *HBV* with $N = 8$ and $k = 2$ (the layered version is shown in the left of Figure 1). The bits of the lowest level represent the bit vector for the set that contains 2, 3, and 6. The bits of the second level (up from the bottom) summarize pairs of bits from the lowest level. Here the first bit in the second level records whether or not either of the values 0 or 1 is in the set (*i.e.*, if either of the first two bits is set in the bottom level). The second bit of the second level records whether or not either of the values 2 or 3 is in the set, and so on. The bit vector at the third level (from the bottom) summarizes pairs of bits from the second level (*i.e.*, groups of four integers from the underlying set). Working up, at the top level a single bit records whether or not the set is empty.

An *HBV* can also be viewed as a tree. For example, if $k = 2$ and N is a power of two, then the *HBV* can be thought of as a full binary tree with N leaves at the lowest level. Each of those leaves corresponds to a number from 0 to $N - 1$ and records whether or not that integer is in the set. Each internal node in the tree will contain a 1 if and only if one of its two children contains a 1. The tree view of an *HBV* with $N = 8$ and $k = 2$ is shown in the right of Figure 1. Relating the two views of a *HBV*, the nodes from a given level of the tree correspond to the bit vector at the same layer as seen in Figure 1.

This chapter introduces *HBV*s. It first considers several applications where the information in the upper-level bit vectors is used to speed up set operations. The chapter then presents a theoretical and an experimental analysis of *HBV*s. The theoretic analysis carefully bounds the constants of the *HBV*s operations. The empirical work compares the running time of the *HBV* with that of simple linked lists and binary search trees.

2 APPLICATIONS

To provide a context for the investigation of *HBV*s, we first consider four applications. In each the *HBV* represents an alternative to simple bit vectors and tree-based structures such as binary search trees, skiplists [5], and other competing data structures [4, 9]. The applications require the following operations over sets of positive integers where the largest possible element is known *a priori*:

create()	make an empty set
insert(n)	add an element to the set
delete(n)	remove an element from the set
contains(n)	determine if an integer is in the set
min()	find the smallest element in the set
succ(n)	find the smallest element in the set that is greater than n

The first of the four applications considered is the conversion of nondeterministic finite autotmata (*NFA*) to deterministic finite automata (*DFA*) [3]. The standard algorithm (see Algorithm 1), first assigns an arbitrary but unique integer label to each state of the *NFA* and then builds sets of states that represent which states the nondeterministic machine could be in. The operations this algorithm performs on the state sets are those given above. For example, min and succ are used to iterate through the state sets in sorted order to check if a state set is equal to a previously generated state set.

In the second application Putonti et al. consider the problem of testing for the presence of certain organisms (*e.g.*, Anthrax) while excluding all DNA subsequences that are present in a selected host (*e.g.*, a benign bacteria) or a background genome (*e.g.*, the human genome) [6]. This is done by identifying all the fixed-length DNA subsequences (k-mers) that occur in all of one set of genomes, but in none of another set of genomes.

The basic idea is to set in correspondence each of the 4^k k-mers and a particular element of a counting array, A, by converting each k-mer character sequence to an index of an element in A. As described by Putonti et al. the algorithm builds a bit vector of every possible k-mer (note that for k=12 there are 4^{12} possible 12-mers). (In contrast traditional blast-based approaches use approximation heuristics and thus may miss certain cases.) The approach then computes matches using a sequence of set unions and intersections. Based on the experiments presented in this chapter, the *HBV* would make a good choice for this identification.

The third application is to non-averaging sets. A non-averaging set is a set of integers that does not contain the average of any pair of its elements (this is equivalent to saying that the set does not contain all three elements of any arithmetic progression $x, x + d, x + 2d$; thus, non-averaging sets are sometimes called 3-free sets). Gasarch et al. survey and give empirical results from several methods of constructing large non-averaging subsets of a fixed range $0, \ \cdots, \ N - 1$ [2]. The best method (due to Behrend [1]) still does not produce sets of optimal density. Post-processing of the sets in order to add elements can be performed using the *HBV*'s operations. Furthermore, the maximum value and density of the resulting sets are within the range for which *HBV*s perform best relative to the alternatives.

Algorithm 1 *NFA* to *DFA* conversion

```
Require: $M = (K, \Sigma, \Delta, s, F)$ is an $\epsilon$-free NFA.
Ensure: $M' = (K', \Sigma, \delta, \{s\}, F')$ is a DFA such that $L(M') = L(M)$.
  Add $\{s\}$ to $K'$.
  repeat
     Choose a state $S$ in $K'$ such that $\delta(S)$ is undetermined.
     $D \leftarrow \emptyset$
     for all $\sigma \in \Sigma$ do                                   ▷ Determine $\delta(S, \sigma)$
        for all $s \in S$ do
           for all $q$ such that $(s, \sigma, q) \in \Delta$ do
              $D \leftarrow D \cup \{q\}$                              ▷ Uses insert
           end for
        end for
     end for
     if $D \notin K'$ then                                           ▷ Can use min and succ
        $K' \leftarrow K \cup \{D\}$
        if $D \cap F \neq \emptyset$ then
           $F' \leftarrow F' \cup \{D\}$
        end if
     end if
     $\delta(S, \sigma) \leftarrow D$                                  ▷ $\delta(S, \sigma)$ is now determined
  until All states in $K'$ have $\delta$ determined
```

The final application is in information retrieval (IR) and exploits a form of lazy allocation. The *HBV*'s bit vectors are usually initialized by setting all the bits to zero. For sparse sets, the initialization time can dominate the total time. Lazy initialization avoids allocating and zeroing tree nodes until they are needed. Thus, the first time a bit is set among a group of k siblings, we allocate the node and then initialize all the other bits in the group to zero. In exchange for this space and initialization-time savings, membership tests can no longer be performed by checking the corresponding bit in the lowest level. Instead, the algorithm must do a top-down traversal of the structure.

Lazy allocation is important for IR applications such as the building of a *concordance* [7, 8]. A concordance records, for a list of keywords $w_1, \cdots, w_n$, which documents in the collection $d_1, \cdots, d_n$ contain which keywords. A bit vector is a natural way to represent this data: for each keyword w_i store a bit vector b_i such that the jth bit of b_i is set if and only if document d_j contains w_i. As concordances are often stored on slow secondary storage units (hard drives or optical disks, for example), the reduced storage space can lead to time savings from the reduced number of I/O operations necessary to read the data.

Typically, the resulting bit vectors will be very sparse; thus, lazily initialized *HBV*s will naturally compress the data, as blocks of all zeros are never allocated. The remaining bits in the higher levels allow the complete structure to be recovered: anywhere a bit vector contains a zero, we know a group of zeros is in the vector at the next lowest level.

3 Hierarchical Bit Vectors

This section first considers the special case of a binary *HBV* (a *HBV* with $k = 2$) before generalizing to a k-ary *HBV*. The discussion of binary *HBV*s first considers two representation choices. It then looks at the definition and complexity of the six *HBV* operations: create, contains, insert, delete, min, and succ. The key to an efficient implementation is the function that determines the next bit to visit as part of the succ operation; thus, careful consideration is given to this function.

3.1 Binary Hierarchical Bit Vectors

Two representations for a binary *HBV* are considered. The first stores the data in a single bit vector. Viewing this bit vector as a tree, the root of the tree is stored in bit zero. The left child of an internal node stored in bit i is stored in bit $2i + 1$; its right child is stored in bit $2i + 2$. The parent of bit i is bit $\lfloor (i-1)/2 \rfloor$.

As an alternative, the structure can be stored in an array of bit vectors having length N, $N/2$, $N/4$, $\cdots$. In the subsequent discussion it is convenient to number the bottom layer as Level 0 and the remaining levels in increasing order as we go up the layers towards the root. Again viewed as a tree, the children of bit i at layer l are then bits $2i$ and $2i + 1$ in layer $i - 1$ (not $2i + 1$ and $2i + 2$ as in the single bit vector case). The parent of bit i of layer l is bit $\lfloor i/2 \rfloor$ at layer $l + 1$. The empirical study presented in Section 4 uses this method because of the simplicity of implementation.

The size of the single bit vector is no more than $3N$. In the case where N is a power of two, it is $2N - 1$. Otherwise, there are wasted bits. For example, the lowest level can include at most $N - 1$ unused (wasted) bits (at N unused bits, the height of the tree can simply be reduced by 1). At higher levels, bits that summarize only unused bits are also wasted bits. Using layers of bit vectors obviates the need to allocate space for unused bits as they always occur at the end of the vector. However, it has to allocate the array to hold the levels, which has size $O(\log_2 N)$.

We now analyze the complexity of the six *HBV* operations: create, contains, insert, delete, min, and succ. The *HBV* create operation takes $O(N)$ time for a set capable of holding the integers $0, \cdots, N - 1$. The contains(n) operation can be performed in $O(1)$ time by simply examining the appropriate bit at the bottom of the tree. The insert operation in a *HBV* is shown in Algorithm 2. In essence, this algorithm first finds the appropriate bit in the vector at the bottom layer. If the bit is not set, it sets it and moves to the next level up. We continue upwards through the layers setting bits until we find a bit that is already set or we come to the root of the tree. Using this algorithm, we can perform an insertion in worst-case $O(\log_2 N)$ time. As the time for insert depends on the number of previous insertions, we note that the total time over a sequence of insertions with no intervening deletions is proportional to the number of bits which must be set, which is $O(N)$ since each bit can be set at most once. The delete operation starts on the lowest level where it clears the selected bit. This bit is part of a pair summarized at the second to last level. The delete operation must check the other summarized bit. If this bit is zero, then the process

Algorithm 2 Inserting an integer in a *HBV*

```
Require: $0 \leq x < N$.
Ensure: $x \in$ HBV
  $l \leftarrow 0$
  $b \leftarrow x$
  while level $l$ is not higher than the highest level do
      if bit $b$ at level $l$ is set then                    ▷ (*)
          STOP
      else
          Set bit $b$ in level $l$                           ▷ (**)
          $l \leftarrow l + 1$
          $b \leftarrow \frac{b}{2}$
      end if
  end while
```

is recursively repeated up one level until the root of the tree is reached. Thus, deletion is worst-case $O(\log_2 N)$ time.

The min and succ operations are used to determine if two *HBV*s represent the same set. This is done by enumerating the two in sorted order. Our goal is to enumerate the elements faster than can be done with a standard bit vector. The bit vector's weakness is that it has to examine each bit in turn; thus, for small sets with large ranges, it must examine long stretches of 0's. We can overcome this by using the additional data stored in the *HBV*. Once we have found one integer in the set, we can use the information in the next-to-last level of the *HBV* to answer the question "is either of the next two integers in the set?" If the answer is no, we use the next level up to answer the question "are any of the next four integers in the set?" Continuing in this way, we can leap over long expanses of emptiness in a few bounds. Once the answer to one of our questions is 'yes', we can go down the identified subtree to find which integer is next.

It is easier to describe the min and succ operations using the tree view of a *HBV*. It is also convenient to define a new operation min(n) that finds the minimum element contained in the subtree rooted at node n. Thus, min() is equivalent to min(root). The succ operation, described below, also uses the min(n) operation.

To perform the min(n) operation on a *HBV* we begin at node n. If there is a zero stored in that bit then the (sub)set is empty and we report that fact. If there is a one, we test the left child. If the left child is one, we visit that node, otherwise we visit the right child. In either case we proceed recursively. The complete algorithm is given as Algorithm 3. This operation clearly takes time proportional to the number of layers, which is $O(\log_2 N)$.

Finally, the succ(x) operation, Algorithm 4, makes use of one more auxiliary function, next(x), which denotes the next bit to consider after considering bit x. Thus, when searching for the next element of a set, the succ operation considers first next(x), then next(next(x)), and so forth until a bit that is set is encountered or the search falls off the side of the tree, in which case x was the largest element in the set. In the following paragraphs several different

Algorithm 3 The min operation for a *HBV*

```
Require: $n$ is a node in the HBV
Ensure: The algorithm terminates with $min$ equal to the index of the leftmost set bit at the
   bottom of the subtree rooted at $n$, or $-1$ if and only if $n$ is not set.
   if $n$ is not set then
      $min \leftarrow -1$
      STOP
   end if
   while $n$ is not a leaf do
      $n \leftarrow$ left($n$)
      if $n$ is not set then
         $n \leftarrow$ sibling($n$)
      end if
   end while
   $min \leftarrow$ the index of node $n$
```

candidate next functions are considered. Some candidates may return an interior node, n, from the tree. In this case, the successor is the smallest element in the tree rooted at n (*i.e.*, min(n)).

The definition of next(x) has a significant impact on the performance of the algorithm. Three possibilities are considered. First, if next(x) = x$+1$ then we are always checking along the bottom of the tree; this corresponds to how we would search in a normal bit vector. A better choice for sparse sets is to move up one or more levels for right children. The second options move up a fixed single level:

$$\text{next(x)} = \begin{cases} \text{x} + 1 & \text{if x is a left child,} \\ \text{parent(x)} + 1 & \text{if x is a right child.} \end{cases}$$

Finally, the third choice allows succ to find the ideal level. Initially we define next(x) as the node on the right branch from the last (lowest) node from which the path from the root to x went left; this will visit nodes in the same order as a pre-order recursive traversal that skips the children of nodes where the bit is not set. Note that for a left child this is x + 1 as it was for Option 2. After some analysis a *cap* will be placed on just how far up the structure to go.

All three next functions result in different answers to the question “when we reach the right edge of a subtree, how far up in the tree do we start searching?” In the first case, we always look at the same level and hence never get out of the bottom level. In the second case we look up at most one level. In the third case we look up as far as possible. Starting from the black node, Figure 2 illustrates these three: going across the bottom (choice A) works well when the set is dense. Going as far up the tree as possible works well when the set is sparse (choice C). Going one level up (choice B) works well for in-between cases.

The remainder of this section demonstrates that the best definition of next is one that dynamically determines which of the latter two options works best for a particular set.

Algorithm 4 The succ operation for a *HBV*

```
Require: $0 \leq x < N$
Ensure: The algorithm terminates with $succ$ equal to the smallest element of the set greater
  than $x$, or -1 is there is no such element.
  $n \leftarrow$ the node containing bit $x$
  while $n$ is not null and is not set do
      $n \leftarrow$ next($n$)
  end while
  if $n$ is null then
      $succ \leftarrow -1$
  else if $n$ is a leaf then
      $succ \leftarrow$ the index of node $n$
  else
      $succ \leftarrow$ min($n$)
  end if
```

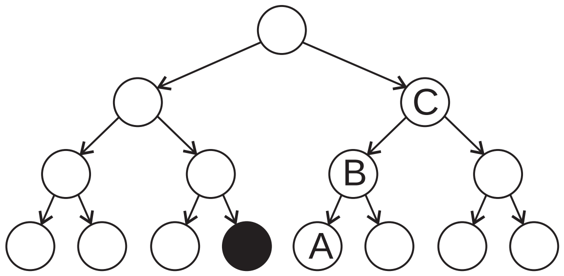

Figure 2: From the black node we can go to A, B, or C.

Consider the choice of going to a node at level l versus going to its left child at level $l-1$. For example, from the black node in Figure 2, this is the choice of going to node C or node B. If we expect the bit at C to be 0 then we should visit it next; doing so will skip its two children altogether. However, if our expectation is incorrect, then upon finding that the node was 1 we will have to visit B next – we would have been better off going directly to B.

To simplify the following analysis, we assume that the elements of the set are chosen randomly and uniformly. Let p denote the probability that an integer is included in the set and $q = 1 - p$ the probability that an integer is not in the set. Thus, for example, the size of the set, n, is expected to be pN. As a further simplification, assume that $N = 2^h$ for some natural number h.

In general, if the node at level l is 0 then we save one visit by going directly to it; that happens with probability q^{2^l}. If that node is 1 we visited one extra node; that happens with probability $1 - q^{2^l}$. Thus, we want to go to the node at level l when that move will, on average, save us work, or when $q^{2^l} \geq 1 - q^{2^l}$. This leads to the sequence of equivalent inequalities

$$\begin{aligned} q^{2^l} &\geq 1 - q^{2^l} \\ q^{2^l} &\geq \frac{1}{2} \\ 2^l \cdot \log_2 q &\geq -1 \\ 2^l &\leq -\frac{1}{\log_2 q} \\ l &\leq -\log_2(-\log_2 q). \end{aligned} \tag{1}$$

If l satisfies the last inequality then it is to our advantage to go to level l instead of level $l-1$; the optimal l is the greatest level l that satisfies that inequality: $l_{opt} = \min(\lfloor -\log_2(-\log_2 q)\rfloor, h)$. Alternately, we can find the densities for which a given level is better than the one beneath it by rewriting the above inequalities as

$$q \geq \left(\frac{1}{2}\right)^{\frac{1}{2^l}}$$

and evaluating for different values of l. For example, for $l = 1$ we get $q \geq \sqrt{\frac{1}{2}}$ and for $l = 2$ we get $q \geq \sqrt[4]{\frac{1}{2}}$.

As we build a set, q will start at 1 and decrease. As q decreases, l decreases. To simplify keeping track of the optimal l (by which we mean keeping down the number of floating point operations needed to compute l), we can first solve for d_{h-1}, the highest density for which traversing level $h-1$ is better than traversing level h. This can be done by taking the original inequality and solving for q when $l = h-1$ and yields $d_{h-1} = 1 - (\frac{1}{2})^{\frac{1}{2^h}}$. When that density is reached, we compute d_{h-2}, which is just $1 - (1 - d_{h-1})^2$.

Based on the proceeding analysis we update Option 3 for the function next such that next(x) determines k, the maximum number of levels it can move up from x. For example, in Figure 3, the values of k for nodes A, B, and C are 1, 2, and 3 respectively. If moving up k levels results in a move above the optimal level l, it moves only as high as level l. Otherwise, it moves up all k levels. This cap on how far up the tree to go complicates the computation of next(x) for right children (for left children we still have next(x) $= x + 1$). Note that it is not the case that every time we move up a level we move to the optimal level: we only move to the optimal level from the rightmost child of a subtree rooted at that level. For example, if the optimal level is Level 2 then from node A we cannot move to the optimal level, but nodes B and C will move to the optimal level (from C we do not move as high as possible because such a move would not be optimal).

When each level is kept in a separate vector, the location of the least significant zero in the binary representation of $x + 1$ gives k. We can then 'shift compute' next(x) by shifting x right by k bits to find the index of x's ancestor within the desired level. In the case where a single bit vector is used for all of the levels, similar bit-level computations can be used. Note that in both cases a lookup table can be used to compute the location of the least significant zero on machines for which there is no appropriate hardware instruction to perform that calculation.

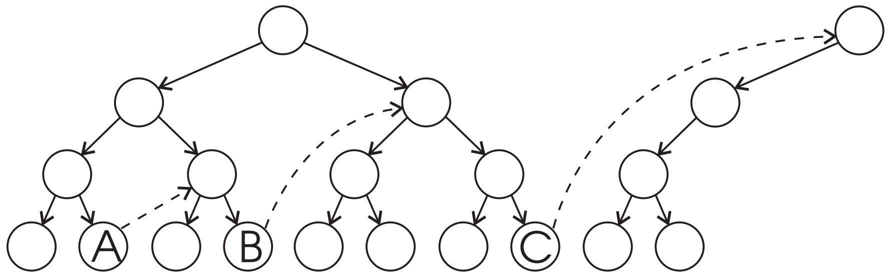

Figure 3: Maximum possible levels to move up in a binary *HBV*.

Having explained the three options used for the function next, we now analyze the running time to enumerate a set using the succ operation. Some intermediate values will be convenient in the ensuing computations: let $C(l)$ denote the expected number of nodes visited in a subtree rooted at level l given that we enter it at its root and let $C'(l)$ denote the expected number of nodes visited in a subtree entered at its level-l root given that the root's parent was 1. Then $C(0) = C'(0) = 1$ and

$$C'(l) = 1 + \frac{1 - q^{2^l}}{1 - q^{2^{l+1}}} \cdot 2C'(l-1) \tag{2}$$

for $l > 0$. In other words, $C'(l)$ is the work done to visit the root plus the probability that the root is non-zero times the work that must be done to visit the children, given that their parent was non-zero. $C(l)$ needs to account for the work done at the root and the work that is done for its two children in the case that the root is non-zero:

$$C(l) = 1 + (1 - q^{2^l}) \cdot 2C'(l-1). \tag{3}$$

If l is the optimal level next should skip to, then an enumeration will visit *every* node at level l. The total work done for a traversal done at level l, denoted by $T(l)$, is therefore the number of nodes at level l times the amount of work done to traverse each level's subtree:

$$T(l) = \frac{N}{2^l} \cdot C(l) = N \cdot \frac{C(l)}{2^l} \tag{4}$$

where $\frac{C(l)}{2^l}$ is the constant of proportionality hidden by the $O(N)$ from before.

To bound $C(l)$ we first bound the q^{2^l} term that appears in both Equations 2 and 3. From Equation 1 we can derive

$$\sqrt{\frac{1}{2}} \geq q^{2^l} \geq \frac{1}{2}$$

for the optimal value of l, where the minimum value is achieved when $-\log_2(-\log_2 q)$ is an integer. We also have

$$\frac{1 - q^{2^l}}{1 - q^{2^{l+1}}} \leq \frac{2}{3},$$

which can be used to find an upper bound on $C'(l)$: substitution into Equation 2 gives

$$\begin{aligned} C'(l) &\leq 1 + \frac{2}{3} \cdot 2C'(l-1) \\ &= 1 + \frac{4}{3}C'(l-1) \\ &= \sum_{i=0} l \left(\frac{4}{3}\right)^i \\ &= 3\left[\left(\frac{4}{3}\right)^{l+1} - 1\right]. \end{aligned}$$

In addition, we have $(1 - q^{2^l}) \leq \frac{1}{2}$, and so we can refine Equation 3:

$$\begin{aligned} C(l) &= 1 + (1 - q^{2^l}) \cdot 2C'(l-1) \\ &\leq 1 + C'(l-1) \\ &\leq 1 + 3\left[\left(\frac{4}{3}\right)^{l+1} - 1\right] \\ &= 3\left(\frac{4}{3}\right)^{l+1} - 2 \end{aligned}$$

.

The constant of proportionality in Equation 4, $\frac{C(l)}{2^l}$, can then also be bounded above by substitution:

$$\begin{aligned} \frac{C(l)}{2^l} &\leq \frac{\left[3\left(\frac{4}{3}\right)^{l+1} - 2\right]}{2^l} \\ &= 3 \cdot \left(\frac{2}{3}\right)^l - \frac{2}{2^l} \\ &= 3 \cdot (2^l)^{\log_2 \frac{2}{3}} - \frac{2}{2^l} \\ &\leq 3 \cdot \left(-\frac{1}{\log_2 q}\right)^{\log_2 \frac{2}{3}} - \frac{2}{-\frac{1}{\log_2 q}} \\ &= 3(-\log_2 q)^{log_2 \frac{3}{2}} + 2\log_2 q. \end{aligned} \tag{5}$$

The first term in this expression is the dominant one.

The following table gives values for the number of nodes visited enumerating the set in sorted order. The first line shows the number when the level used by next is capped based on Equation 5. The values when there is no cap on how high up in the tree the next function goes are listed on the second line for comparison. These figures can be compared with the work done by an ordinary bit vector, which is always $1.00N$. Note that because the fraction of nodes tends to zero along with p, there must be a density below which the running time to enumerate the set in sorted order is lower for a *HBV* than for a standard bit vector.

$p =$	0.000135	0.00540	0.0214	0.0830	0.293
cap	$0.0165N$	$0.0772N$	$0.223N$	$0.555N$	$1.000N$
no cap	$0.0311N$	$0.0929N$	$0.251N$	$0.583N$	$1.077N$

As illustrated above, for low density sets, *HBV*s outperform standard bit vectors because they can enumerate a set in sorted order faster. For higher density sets, *HBV*s are better than tree structures because they can perform a collection of inserts faster. The time taken for all insertions (assuming no intervening deletes) is proportional to the number of times the two marked lines of Algorithm 2 are executed. The total number of times those lines are executed is equal to the number of bits that are set in *HBV* plus the number of times the insert operation is used. Each of the 2^h leaves of the tree is set with probability $p = 1 - q$. Each of the 2^{h-1} nodes at the second level from the bottom is set with probability $1 - q^2$. Proceeding in this manner, we can compute $I(h)$, the expected number of bits that are set in a tree of height h:

$$I(h) = \sum_{i=0}^{h} 2^{h-i}(1 - q^{2^i}).$$

This is the number of times the line in Algorithm 2 marked with (**) is executed. The line marked (*) is executed once for each insertion; if all the insertions are unique then (*) is executed $n = pN = (1-q)N$ times. The total $I(h)+n$ is a measure of the work done by *all* insertions together; the cost per insertion is $\frac{I(h)}{n} + 1$. A table of this value for different combinations of N and p is given in Figure 4.

Note that for a fixed density p, the average work per insertion approaches a constant and so the total work for n insertions is $O(n)$. Also under this assumption, the work to enumerate the set in sorted order is $O(N) = O(\frac{n}{p}) = O(n)$. Therefore, for fixed p, an *HBV* performs n distinct insertions followed by a enumeration in $O(n)$ time. On the other hand, a binary search tree will perform $O(n \log_2 n)$ work to do n distinct insertions followed by $O(n)$ work to enumerate the set in order for a total of $O(n \log_2 n)$. Therefore, for a fixed density, there is always an N large enough so that an *HBV* will outperform a binary search tree or similar structure. For example, the empirical data presented in Section 4 show that the break-even point for $p = \frac{1}{1024}$ is around $N = 2^{26}$. Larger p yield lower break-even points: for $p = \frac{1}{4}$ it is around $N = 1024$.

However, as p increases, the advantage the *HBV* has over the bit vector gets smaller and smaller until it disappears completely. Tests have shown that for $p \leq \frac{1}{65536}$ the *HBV* outperforms an ordinary bit vector. However, for such sparse sets the range must be extremely high for the *HBV* to outperform a binary tree.

3.2 k-ary Hierarchical Bit Vectors

This section generalizes the 2-ary *HBV*s from the previous section to values of k other than 2. The high performance of the ordinary bit vector comes from the fact that no processor works one bit at a time. If we want to list the elements stored in a bit vector in order using a 32-bit processor, we can break the vector into words of 32 bits each and examine each word

	p (the probability that an integer is included in the set)				
$N(=2^h)$	p =0.0020	p =0.0078	p =0.0312	p =0.1250	p =0.5000
2	3.00	3.00	2.98	2.94	2.75
4	3.99	3.98	3.94	3.77	3.22
8	4.99	4.96	4.84	4.42	3.47
16	5.97	5.90	5.63	4.86	3.59
32	6.94	6.79	6.27	5.11	3.66
64	7.89	7.58	6.70	5.23	3.69
128	8.77	8.21	6.95	5.30	3.70
256	9.56	8.64	7.08	5.33	3.71
512	10.2	8.89	7.14	5.34	3.71
1024	10.6	9.02	7.17	5.35	3.72
2048	10.9	9.08	7.18	5.36	3.72
4096	11.0	9.11	7.19	5.36	3.72
8192	11.1	9.12	7.20	5.36	3.72
16384	11.1	9.13	7.20	5.36	3.72
32768	11.1	9.14	7.20	5.36	3.72
65536	11.1	9.14	7.20	5.36	3.72
131072	11.1	9.14	7.20	5.36	3.72
262144	11.1	9.14	7.20	5.36	3.72
524288	11.1	9.14	7.20	5.36	3.72

Figure 4: Insertion Cost

using one instruction. If a word is zero then we can ignore it. Only if it is non-zero do we have to slow down and check it bit by bit. We use the same strategy to speed up the *HBV*.

A k-ary *HBV* (k-*HBV* for short) is similar to a binary *HBV*, but each node in the tree has k children all stored in the same k-bit word. Insertions and membership tests can be done in much the same way as in the binary *HBV*. The next operation can be reworked to take advantage of the power of microprocessors to examine more than one bit at a time: when finding the next integer in the set after m, we first examine the rest of the bits in m's word. If one is non-zero, we find the first non-zero bit, often using a single CPU instruction, for example BSR ("bit scan reverse") on the x86 family of processors. If all the other bits were zero, we choose a word in a higher level (the sibling of an ancestor of m's word) and examine it, continuing in this manner until we find a non-zero bit. Once found, we can go down the tree to find the smallest non-zero descendant.

For example, in the 4-ary *HBV* shown in Figure 5, to find succ(0) we would mask the most significant bit of the first word at the bottom level to get the bit pattern 0001. The BSR instruction tells us that the first non-zero bit is bit number 3, so we know that succ$(0) = 3$. To compute succ(3), we first notice that 3 was the last bit of its word, so we go directly to that word's parent, the first word of the middle layer of the figure. We have already

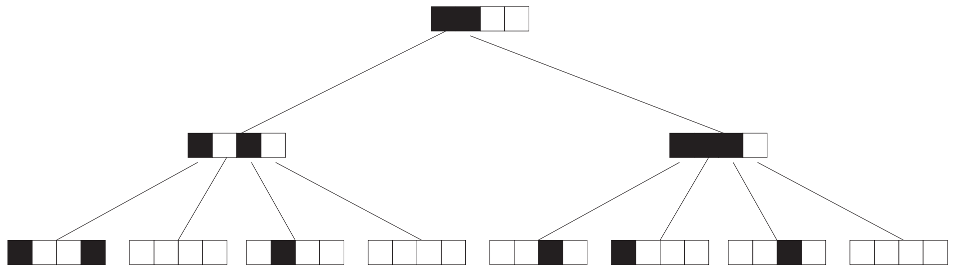

Figure 5: An *4*-HBV representing $\{0, 3, 9, 18, 20, 26\}$.

examined that word's leftmost child, so we again mask off the most significant bit to get 0010. BSR tells us that the first non-zero bit is number 2, so we next examine the third child where we find the bit pattern 0100. BSR now tells us that the first bit set is bit number 1, so $\text{succ}(3) = 1 + 2 * 4 = 9$, for bit number 1 in word number 2 at the lowest level.

With the k-*HBV* we face the same choice of the highest level used to traverse the structure during a sorted enumeration as we did for the binary *HBV*. In Figure 5, if we traverse at the middle level we will examine the first word at the middle level and discover that its first bit is set, forcing us to examine the first word on the second level; we discover that it's first bit is set and then examine the last three bits, finding that the last is set; at that point we return to the middle level and check the remaining three bits of the first word. By continuing in that manner we will examine 17 words or parts of words. By simply going along the bottom layer we would have examined only 13. In this case the set is dense enough that a standard bit vector would outperform a *HBV*.

In general, whenever we probe a word at any level of a k-*HBV*, we will examine the entire word once and will examine parts of the word for each one we find in bits 0 through $k - 2$. So if we examine a word at level l that has m non-zero children among its first $k - 1$ children, we should expect to examine that word $m + 1$ times. We will also have to examine those non-zero children but we will not examine the children with all zero bits. On the other hand, if we bypass the level-l word and instead examine all of its children, we will do the same work that we did before for its non-zero children plus one examination each for the children with all zero bits.

The expected benefit of traversing level l instead of $l - 1$ is then the expected number of zero words at level $l - 1$ minus the expected number of non-zero words among the first $k - 1$ children minus 1. The probability that a word at level $l - 1$ is zero is $(1 - p)^{k^l}$ so the expected number of empty children is $k \cdot (1 - p)^{k^l}$. The probability that a word at level $l - 1$ is non-zero is $1 - (1 - p)^{k^l}$, so the expected number of those among the first $k - 1$ children is $(k - 1) \cdot (1 - (1 - p)^{k^l})$. The expected benefit is then

$$k \cdot (1 - p)^{k^l} - (k - 1) \cdot (1 - (1 - p)^{k^l}) - 1.$$

As long as the above value is non-negative it is better (or at least just as good) to traverse level l instead of level $l-1$. After some algebra, we conclude that traversing level l is better than traversing level $l-1$ when

$$(1-p)^{k^l} \geq \frac{k}{2k-1}.$$

By plugging in values for k and l, we can find the densities that are good for traversing at any level. For $k = 32$ and $l = 1$, we find that $p \leq 1 - \sqrt[32]{\frac{32}{63}} \approx \frac{1}{48}$, which means Option 2 for the function next is preferred over Option 1 as traversing level 1 we examine fewer words than an ordinary bit vector (Option 1) for sets that are less than $\frac{1}{48}$ full. For $k = 32$ and $l = 2$ we have $p \leq 1 - \sqrt[1024]{\frac{32}{63}} \approx \frac{1}{1512}$ and for $k = 32$ and $l = 3$ we get $p \leq 1 - \sqrt[32768]{\frac{32}{63}} \approx \frac{1}{48374}$; however, at this point the binary search trees has a decided advantage for commonly used values of N.

4 Empirical Analysis

The three data structures: bit vectors, *HBV*s, and binary search trees were each implemented and used in an empirical evaluation. This complements the analysis from the previous section, which counted the number of words examined to enumerate a set in sorted order, but ignored the associated constants and the work that has to be done for other operations. The experiments confirm that tree structures are superior for small sparse sets, bit vectors are superior for dense sets, and *HBV*s are the best choice for large but sparse sets.

We ran the three implementations using randomly generated sets with various values of N and p on an AMD Athlon 64. Each test consists of constructing an empty set, inserting each integer with probability p (so the expected number of elements in the set is pN), and finally outputting the elements of the set in sorted order. The order of the insertions is randomized to avoid the worst-case $O(n)$ behavior of the binary search tree for this operation. Each test run is intended to reflect the usage pattern of *NFA* determinization, where state sets are created, populated, and then output in sorted order to allow comparisons to other sets. We include only one enumeration because in the determinization algorithm, the insertions are all performed before any enumerations; thus, the result from the the first can be stored and used for subsequent operations. A model of the sequence of operations for post-processing non-averaging sets would be similar but would include contains and delete operations as well.

Three *HBV* design decisions warrant mention. First, given the hardware we used, an appropriate value for k was 32. Second, to save initialization time the *HBV* implementation only initialized the top layer (to zero). The first time a bit is set (at internal nodes and leaves) among a group of k siblings, we initialize the other bits in its group to zero. Testing whether we are setting the first bit in a group of k siblings can be determined by checking whether we have just changed the parent from 0 to 1. The sorted-order enumeration algorithm does not need to be changed because it will never check bits that have not been initialized.

Membership tests can not be performed by checking the corresponding bit in the lower lever, however. Instead, we must do a top-down traversal of the structure.

The third design decision follows from the theoretical result that levels above the bottom three would only be useful for very sparse sets ($p < \frac{1}{48374}$) where binary search trees have a decided advantage. This is because above this level we do not expect to encounter many zero bits; thus, the function next should never choose to traverse the tree above this level. Retaining at most the bottom three levels pragmatically means either retaining 3 or just 2 (as retaining just 1 would find the *HBV* degenerate into a simple bit vector). For the experiment, rather than create one structure that adapts to different densities, we chose a fixed cutoff. Making the structures non-adaptive significantly reduces the overhead of the operations.

Two different fixed cutoff values were experimented with: a 2-layer 32-ary *HBV* (essentially implementing Option 2 for the function next) and a 3-layer 32-ary *HBV* (essentially implementing a limited version of Option 3). The 2-layer version maintains only the bottom two levels; thus, it assumes that all bits at higher levels are always 1. The 3-layer version maintains the bottom 3 levels. Both choose the topmost retained layer as the optimal layer for the purposes of the next operation.

Comparing the simple bit vector and the two *HBV* implementation empirically (see Figure 6), the data reveal that the 2-level 32-ary *HBV* can enumerate in sorted order in less time than an ordinary bit vector for sets sparser than $p = 2^{-9}$ and the 3-layer 32-ary *HBV* enumerates in sorted order in less time than the 2-layer 32-ary *HBV* for sets sparser than $p = 2^{-14}$. However, over the range of values for N we used during our experiment, densities lower than $p = 2^{-14}$ give a clear advantage to tree structures. Thus, in the remainder of this section only the 2-$layer$ 32-ary *HBV* is considered.

The results of the experiment are summarized graphically in Figure 7. The x-axis of the figure shows the range of values while the y-axis shows the number of elements inserted. Both use a logarithmic scale. The three bands in the figure follow the prediction of the analysis done in Section 3 with the binary search tree being fastest for small sparse sets, the bit vector being fastest for dense sets, and the *HBV* begin fastest in the middle range.

As evident in the figure, the boundaries are not straight lines, but show two anomalies (regions where the results appear to 'backtrack'). The first anomaly occurs at $(N, n) = (2^{17}, 2^5)$ where the *HBV* is faster even though the binary search tree is faster for both $N = 2^{16}$ and $N = 2^{18}$. The second anomaly occurs at $n = 2^{17}$ where bit vectors perform better for $N = 2^{22}$ and $N = 2^{25}$ but not $N = 2^{23}$, $N = 2^{24}$, or $N = 2^{26}$. Both appear to be cache effects. For example, at $N = 2^{22}$ the bit vector would occupy all of the AMD's 512KB L2 cache, yet layer 1 of the *HBV* will fit in the 64KB L1 cache for $N \leq 2^{24}$.

To investigate this hypothesis, the experiment was repeated using an Intel Core 2, which has a larger L2 cache, but a smaller L1 cache. The data, shown in Figure 8, supports the hypothesis: the first anomaly moves to the left along with the decrease in the Core 2's L1 cache size, while the second anomaly moves to the right along with the increase in the L2 cache size.

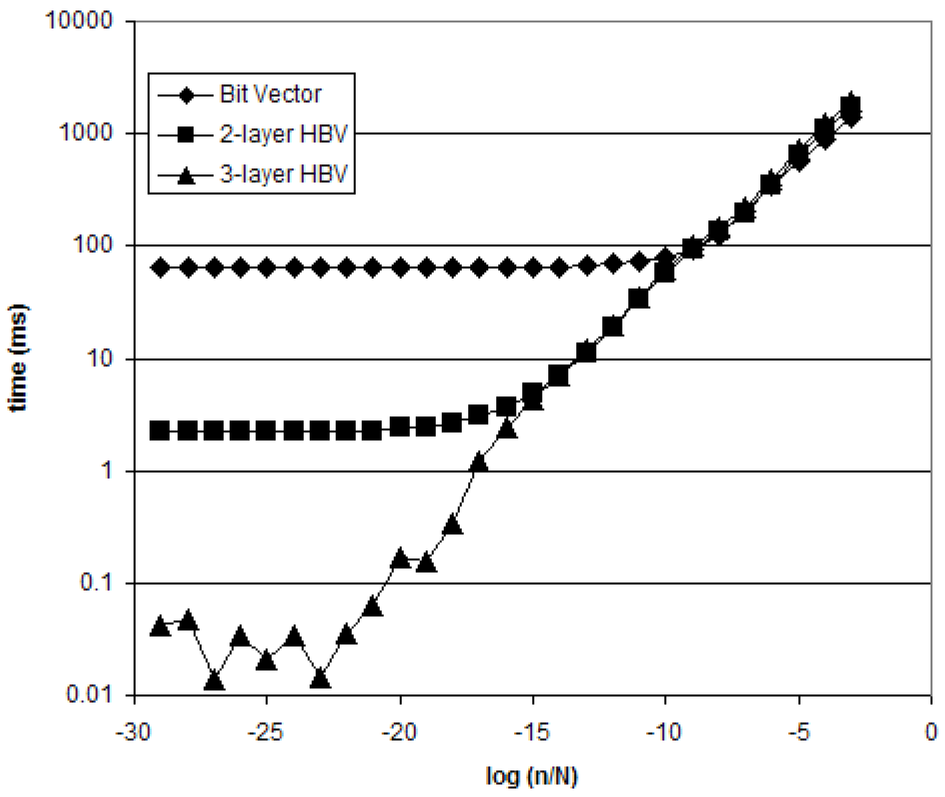

Figure 6: Time to enumerate in sorted order versus density.

5 Conclusion

If it is known ahead of time that an application will make use of sparse sets and will have to perform the succ operation on those sets, then the 32-ary *HBV* can be put to good use. The more times the succ operation is used, the greater the *HBV*'s advantage is over the standard bit vector. The larger the set or the more times the contains operation is used, the greater the *HBV*'s advantage over the binary search tree. Applications such as non-averaging set post-processing where it is possible to determine the density of the sets used, the *HBV* supports a more time-efficient implementation.

References

[1] F. Behrend. On sets of integers which contain no three in arithmetic progression. *Proc. of the National Academy of Science (USA)*, 23:331–332, 1946.

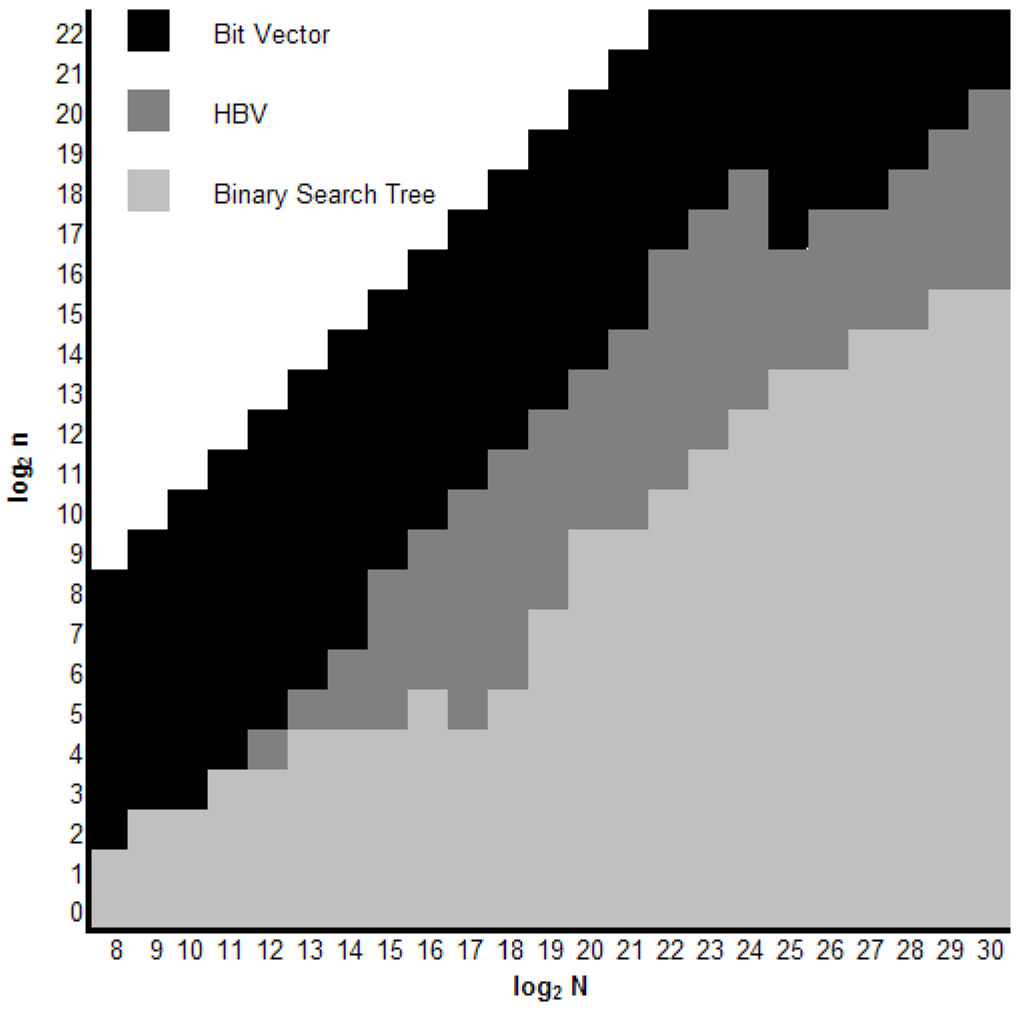

Figure 7: Comparison of three data structures (AMD Athlon 64 Architecture)

[2] W. Gasarch, J. Glenn, and C. Kruskal. Finding large 3-free sets I: the small n case. *Journal of Computer and System Sciences*, (to appear).

[3] James Glenn and William I. Gasarch. Implementing ws1s via finite automata: Performance issues. In Derick Wood and Sheng Yu, editors, *Workshop on Implementing Automata*, volume 1436 of *Lecture Notes in Computer Science*, pages 75–86. Springer, 1997.

[4] M. Korda and R. Raman. An experimental evaluation of hybrid data structures for searching. In *Proceedings of the 3rd International Workshop in Algorithm Engineering*, Lecture Notes in Computer Science 1668, pages 213–227. Springer Verlag, 1999.

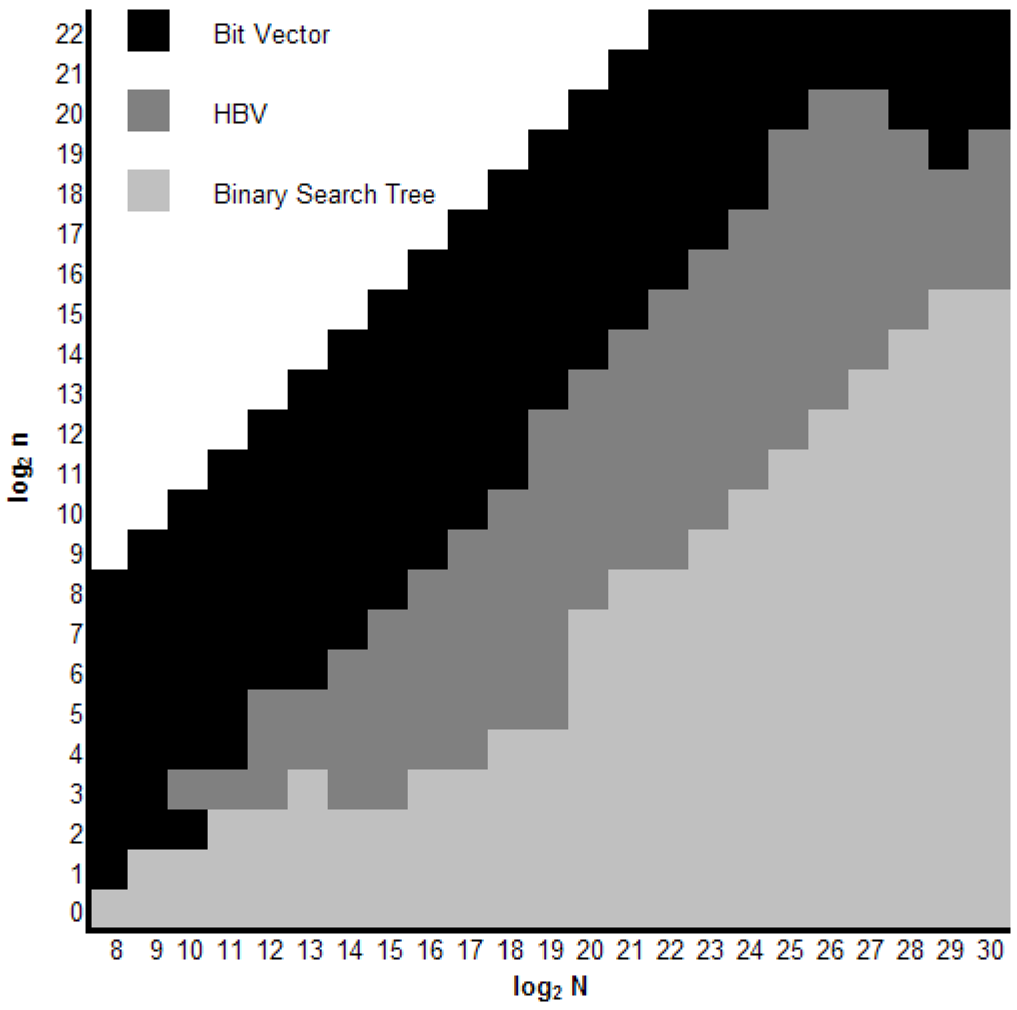

Figure 8: Comparison of three data structures (Intel Core 2 Architecture)

[5] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990.

[6] C. Putonti, S. Chumakov, R. Mitra, G. Fox, R. Willso, and Y. Fofanov. Human-blind probes and primers for dengue virus identification – exhaustive analysis of subsequences present in the human and 83 dengue genome sequences. *Federation of European Biochemical Societies (FEBS)*, 273, Feb 2006.

[7] O. Vallarino. On the use of bit-maps for multiple key retrieval. *SIGPLAN Notices, Special Issue*, II:108–114, 1976.

[8] H. Wedekind and T. Härder. *Datenbanksysteme II*. B.-I. Wissenschaftsverlag, 1976.

[9] D. E. Willard. New trie data structures which support very fast search operations. *Journal of Computer and System Sciences*, 28(3):279–294, 1984.