Optimal lower bounds for rank and select indexes

Alexander Golynski

David R. Cheriton School of Computer Science, University of Waterloo
agolynski@cs.uwaterloo.ca
Technical report CS-2006-03, Version: February 10, 2006

Abstract. We develop a new lower bound technique for data structures. We show an optimal $\Omega(n \lg \lg n / \lg n)$ space lower bounds for storing an index that allows to implement rank and select queries on a bit vector B provided that B is stored explicitly. These results improve upon [Miltersen, SODA'05]. We show $\Omega((m/t) \lg t)$ lower bounds for storing rank/select index in the case where B has m 1-bits in it (e.g. low 0-th entropy) and the algorithm is allowed to probe t bits of B. We simplify select index given in [Raman *et al.*, SODA'02] and show how to implement both rank and select queries with an index of size $(1+o(1))(n \lg \lg n / \lg n) + O(n / \lg n)$ (i.e. we give an explicit constant for storage) in the RAM model with word size $\lg n$.

1 Introduction

The term *succinct data structure* was first used by Jacobson in [2], where he defined and proposed a solution to the following problem of implementing rank and select queries. We are given a bit vector B of length n. The goal is to represent B in such a way that rank and select queries about B can be answered efficiently. Query $rank_B(i)$ returns the number of 1 in B before position i, and $select_B(i)$ query returns the position of the i-th occurrence of 1 in B. We require that the representation should be succinct, that is, the amount of space S it occupies is close to the information-theoretic minimum, namely $S = n + o(n)$ in the case of bit vectors of length n. We consider this problem in the RAM model with word size $w = \Theta(\lg n)$. Jacobson proposed a data structure to perform rank queries that uses $n + O(n \lg \lg n / \lg n)$ bits of space and requires only $O(1)$ time to compute the answer. His implementation of select queries requires $O(\lg n)$ bit accesses, but it does not take advantage of word parallelism and runs in time that is more than a constant in RAM model.

It was subsequently improved by Clark [1], Munro *et al.* [4, 5], and Raman *et al.* [6]. The index proposed by Raman *et al.* [6] occupies $O(n \lg \lg n / \lg n)$ bits, and the select query is implemented in $O(1)$ time. All these data structures belong to a class of *indexing data structures*. An indexing data structure stores data in "raw form" (i.e. B is stored

explicitly) plus a small index I to facilitate implementation of queries, such as rank and select. We denote the size of the index by r.

Miltersen [3] showed that any indexing data structure that allows $O(1)$ time implementation of rank (select) queries must use an index of size at least $\Omega(n \lg\lg n/\lg n)$ bits (respectively $\Omega(n/\lg n)$ bits). The purpose of this paper is to develop a new technique for showing lower bounds for indexing data structures. This technique allows to improve lower bounds of Miltersen [3] for both rank and select problems to match the corresponding upper bounds. We show that any algorithm that performs rank or select queries and uses $O(\lg n)$ bit accesses to a bit vector B, plus unlimited number of accesses to an index I, and is not limited in computation time, must use an index of size at least $\Omega(n \lg\lg n/\lg n)$. Hence,

(i) in the case of select queries, we improve his lower bound and show the optimal one;

(ii) in the case of rank queries, we show optimal lower bound, but in a more general setting.

We also consider the case where the number of 1-bits in a bit vector B is some given number m (we call it *cardinality*). In this setting, for both rank and select problems, we prove a theorem that any algorithm that uses t bit probes to B to compute the answer must use an index of size at least $(m/t)\lg t$. In particular this lower bound is optimal for bit vectors of constant 0-th order entropy. This theorem also yields strong lower bounds in the case $m > n/\lg\lg n$.

We also give an implementation of select query that is simpler than the one proposed by Raman *et al.* [6]. We also give an index that allows to implement both rank and select queries in $O(1)$ time and uses space $(1+o(1))(n\lg\lg n/\lg n)+O(n/\lg n)$. Thus, we give an explicit constant in front of the leading term $n\lg\lg n/\lg n$. This index is simple and space efficient, and it might of interest to practitioners.

This paper is organized as follows. In the section 2, we give an implementation for rank and select queries. In the section 3, we prove lower bounds for rank and select indexes. In the section 4, we consider the case of bit vectors with given cardinality.

2 Upper Bounds

In this section, we will simplify the result of Raman *et. al* [6] that gives an optimal index for the select query of size $O(n\lg\lg n/\lg n)$. [1] Then we will construct an optimal index for rank query of size $(1+o(1))(n\lg\lg n/\lg n)+O(n/\lg n)$. A similar result was obtained by Jacobson [2]; however we implement both rank and select index simultaneously, such that the space used is just $n+(1+o(1))(n\lg\lg n/\lg n)+O(n/\lg n)$. Both of these indexes share a component of size $(1+o(1))(n\lg\lg n/\lg n)$ that we call a *count index*. The count index is constructed as follows: we split our bit string B into *chunks* of size $\lg n - 3\lg\lg n$. Then we store

[1] In this section, we assume that word size $w = \lg n$

the number of 1-bits in each chunk (we call it *cardinality of a chunk*) in equally spaced fields of size $\lg\lg n$ for a total of $n\lg\lg n/(\lg n - 3\lg\lg n) = (1+o(1))n\lg\lg n/\lg n$ bits.

2.1 Optimal Select Index

In this subsection, we describe a new simplified select index that uses count index plus an additional $O(n/\lg n)$ bits. Let B be the bit vector of length n. Let $S_1 = (\lg n)^2$. We store the locations of each (iS_1)-th occurrence of 1-bit in B, for each $1 \le i \le n/S_1$. This takes $O(n/\lg n)$ bits in total. We call regions from position select(iS_1) to position select$((i+1)S_1)-1$ *upper blocks*. To perform select$_B(i)$, we first compute $j = \lfloor i/S_1 \rfloor$ the number of the upper block that the i-th bit is in, so that

$$\text{select}_B(i) = \text{select}_B(jS_1) + \text{select}_{UB_i}(i \bmod S_1)$$

where select$_{UB_i}$ denotes the select query with respect to the i-th upper block. We call such an operation *reduction from cardinality n to S_1*. Now we need to implement the select query for upper blocks. We call an upper block *sparse* if its length is at least $(\lg n)^4$. For a sparse block, we can just explicitly write all answers for all possible select queries, this will use at most $(\lg n)^3$ bits, so that we use $1/\lg n$ bits for an index per each bit in the original bit vector B. Hence we use at most $n/\lg n$ for all sparse upper blocks.

Let us consider a non-sparse upper block. It is a bit vector of cardinality S_1 and length at most $(\lg n)^4$. Thus, it takes $O(\lg\lg n)$ bits to encode a pointer within such a block. We perform cardinality reduction from S_1 to $S_2 = \lg n\lg\lg n$. As above, we introduce *middle blocks*, each having cardinality S_2. That is, encode every (iS_2)-th occurrence of 1-bit in an upper block. This information occupies $O(\lg\lg n \cdot \lg n/\lg\lg n) = O(\lg n)$ bits for an upper block of length at least $(\lg n)^2$, so that we use $1/\lg n$ bits for index per one bit from B, for a total of at most $O(n/\lg n)$ bits. We call a middle block *sparse* if it has length more than $(\lg n\lg\lg n)^2$. If a middle block is sparse, then we can explicitly write positions of all occurrences of 1-bits in it, this uses at most $\lg n(\lg\lg n)^2$ bits (we use at most $1/\lg n$ indexing bits per one original bit). We call a middle block *dense* if its length is at most $\frac{(\lg n)^2}{4\lg\lg n}$.

If a middle block is neither sparse nor dense, then use cardinality reduction from S_2 to $S_3 = (\lg\lg n)^3$. Call the resulting blocks of cardinality S_3 *lower blocks*. That is, store every (iS_3)-th occurrence of 1 in a middle block. This uses $\lg n/\lg\lg n$ bits per block of length at least $\frac{(\lg n)^2}{2\lg\lg n}$, e.g. we use $1/\lg n$ indexing bits per one bit from B. We say that lower block is *sparse* if it has length at least $\lg n(\lg\lg n)^4$ and *dense* otherwise. If a lower block is sparse, then we can explicitly encode all 1-bit occurrences in it.

It remains to implement select query for dense middle and lower blocks. Consider, for example, a dense middle block MB and implement select$_{MB}(i)$ on it. We first assume that MB is aligned with chunks, i.e. its starting (ending) position coincide with starting (ending) position of some chunk

(chunks are of the size $\lg n - 3 \lg \lg n$). Recall that the length of MB is at most $(\lg n)^2/4 \lg \lg n$, so that the part P of the count index that covers block MB (i.e. P encodes cardinality of each chunk inside MB) is of the size at most $(\lg n)/2$. Hence, we can read P in one word and perform a lookup to a table T to compute the number of the chunk where i-th 1-bit of MB is located. Table T is of size at most $\sqrt{n} \lg n \lg \lg n \times \lg \lg n$, and it stores for each possible choice of P and for each $j = O(\lg n \lg \lg n)$ the number of the chunk where j-th occurrence of 1 is located (denote the corresponding chunk by C), and the rank of that occurrence inside C (denote it by p). Now we can compute $\text{select}_{MB}(j)$ by reading chunk C and performing $\text{select}_C(p)$ using a lookup to a table Q. Table Q is of size at most $O(2^{\lg n - 3 \lg \lg n} \cdot \lg n \times \lg \lg n) = O(n/\lg n)$, and it stores for each possible chunk C and for each position k the result of $\text{select}_C(k)$. The case where MB is not aligned with chunks can be resolved by counting number of 1-bits in the first chunk that partially belongs to MB (e.g. a lookup to a table that computes rank within a chunk, we discuss this table later in the next subsection) and adjusting j accordingly. Clearly, select query for the case of dense lower blocks can be implemented in the same way.

2.2 Optimal Rank Index

In this subsection, we show how to design rank index using count index and additional $O(n/\lg n)$ bits.

We divide the bit vector B into equally sized *upper blocks* of size $S_1 = (\lg n)^2$ bits each. For each upper block, we write the rank of the position preceding its first position ($rank_B(0) = 0$). This information uses $O(n/\lg n)$ bits total. Now we can compute $rank_B(i)$ as follows: first we compute $j = \lfloor i/S_1 \rfloor$, the number of the upper block that contains i-th bit of B (denote the upper chunk by UC), so that

$$rank_B(i) = \text{rank}_B(jS_1 - 1) + rank_{UC}(i \bmod S_1)$$

We call such an operation a *length reduction* from n to S_1. Then we perform another length reduction from S_1 to $S_2 = \lg n \lg \lg n$. We call the corresponding blocks of length S_2 *middle blocks*. It takes $\lg n$ bits per an upper block of length $(\lg n)^2$ to describe ranks of the starting positions of middle blocks (each rank uses $\lg \lg n$ bits), so that we use $1/\lg n$ bits for index per one bit of B. Without loss of generality, we can assume that middle blocks are always aligned with chunks. Let MB be a middle block, we implement $\text{rank}_{MB}(i)$ as follows. Let $j = O(\lg \lg n)$ be the number of the chunk (denote it by C) that contains the i-th bit of MB, $j = \lfloor i/S_3 \rfloor$, where $S_3 = \lg n - 3 \lg \lg n$ denotes the length of a chunk. One middle block of size S_2 corresponds to a part P of counting index of size at most $O(\lg \lg n)^2$ bits, so that we can read it in one word and use one lookup to a table T to compute $\text{rank}_{MB}(jS_3 - 1)$. Table T is of size $(\lg n)^{O(1)} \lg \lg \lg n \times \lg \lg n$, and it stores $\text{rank}_{MB}(jS_3 - 1)$ for each possible part P and chunk number j. Thus,

$$\text{rank}_{MB}(i) = \text{rank}_{MB}(jS_3 - 1) + \text{rank}_C(j \bmod S_3)$$

and the latter rank can be also computed by one lookup to the table Q. Recall that table Q of size $O(n/\lg n)$ stores for each possible chunk C and for each position k the result of $\text{select}_C(k)$.

3 Lower Bounds

In this section, we prove a lower bound on the index size for any algorithm that has unlimited access to an index I, has unlimited computation, and is allowed to perform at most $O(\lg n)$ bit probes to the bit vector B. Note that this setting is general enough; in particular, it covers $O(1)$ RAM algorithms.

3.1 Rank Index

In this subsection, we give an argument that techniques from Miltersen [3] does not allow to obtain

$$r = \Omega(n \lg n / \lg \lg n) \tag{1}$$

in the case (i) where we can perform $O(\lg n)$ bit probes to the bit vector B; although in the case (ii) where only $O(1)$ word probes are allowed his lower bound (1) is optimal. Next, we develop a new combinatorial technique and obtain (1) bound for the case (i).
Miltersen [3] showed that the rank index has to be of size r, such that

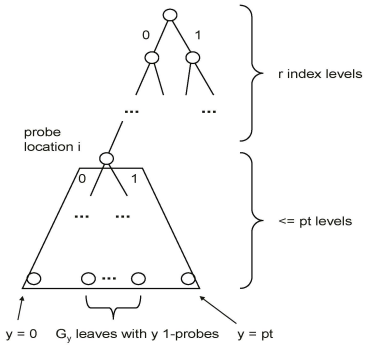
$$2(2r + \lg(w+1))tw \geq n \lg(w+1)$$

where w denotes the word size, t denotes the number of word probes, and r is the size of an index. In the case $w = 1$ and $t = O(\lg n)$, his lower bound only gives $r = \Omega(n/\lg n)$. Miltersen reduces (i) a set of $n/\lg n$ independent problems: problem i is to compute n_i, the number of 1's in the region $[i \lg n, (i+1)\lg n]$ modulo 2 to (ii) the problem of computing rank. He shows $r = \Omega(n \lg n / \lg \lg n)$ lower bound (1) for (i) when $w = \Theta(\lg n)$. However for the case $w = 2$, it suffices to store all the $n \lg n$ answer bits as the index, so that no bit probes are needed to solve (i). One can try to generalize the problem and compute n_i's modulo $\lg n$ but using $O(\lg n)$ bit probes instead of $O(1)$ word probes. In his proof, he encodes a number $i \in [\lg \lg n]$ [2] using unary representation $1^i 0^{\lg n - i}$. In the RAM model with word of size $\lg n$, it takes one probe to access i, however we can recognize i using binary search in $\lg \lg n$ bit probes, so that (i) can be solved in $\lg \lg n$ bit probes and without any index. One can also try to "shuffle" bits in unary representation to disallow such binary searches, however it is not clear whether such a proof can be completed.
Now we describe a different technique and use it to show stronger lower bounds on indexes. Fix the mapping between bit vectors B and indexes I and fix an algorithm that performs the rank query (i.e. it computes

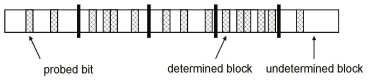
[2] [n] denotes the set {0, 1, . . . , n-1}

$\text{rank}_B(p)$ for a given p). As we mentioned before, an algorithm is allowed to perform unlimited number of bit probes to I and has unlimited computation power; we only limit the number of bit probes it can perform to the bit vector B. Let us fix the number of bit probes $t = f \lg n$ for some constant $f > 0$. We split the bit vector B into p blocks of size $k = t + \lg n$ each. Let n_i be the number of 1-bits in the i-th block, we call n_i the *cardinality* of i-th block. For each block i, $1 \leq i \leq p$, we simulate the rank query on the last position of the i-th block, $s_i = \text{rank}_B(ik - 1)$, so that $n_i = s_{i+1} - s_i$. Note that we will have at least $n - pt = p \lg n = \Omega(n)$ unprobed bits after the computation is complete. Now we will construct a binary *choices tree*. The first r levels correspond to all possible choices of index. At each node at depth r of the tree constructed so far, we will attach the decision tree of the computation that rank algorithm performed for the query $\text{rank}_B(k - 1)$ when index I is fixed. The nodes are labeled by the positions in the bit vector that algorithm probes and two outgoing edges are labeled 0 or 1 depending on the outcome of the probe; we call the corresponding probe a 0-*probe* or 1-*probe* respectively. At each leaf the previously constructed tree, we attach the decision tree for $\text{rank}_B(2k-1)$ and so on. Thus, the height of the tree is at most $r+tp$. If the computation probes the same bit twice (even if the previous probe was performed for a different rank query), we do not create a binary node for the second and latter probes; instead we use the result of the first probe to the bit. At the leaves of the tree all block cardinalities n_i are computed. Let us fix a leaf x, we call a bit vector B *compatible* with x iff: (1) index (i.e. first r nodes) on the root to leaf path correspond to the B's index; and (2) remaining nodes on the root to leaf path correspond to the choices made by the computation described above.

Let us bound the number $C(x)$ of bit strings B that are compatible with a given leaf x (in what follows we will use C to denote $C(x)$).



Choices tree. Leaves could be at different levels. (Notation G_y will be defined in the section 4)



Bit vector at a leaf x. Number of 1-bits in each block is determined by the computation along root to leaf path.

Let u_i be the number of unprobed bits in the block i, so that $u_i \le k$ and

$$\sum_{i=1}^{p} u_i = U$$

where U is the total number of unprobed bits. At a given leaf, we have computed all n_i's, and hence the sum of all unprobed bits (denote it by v_i) in the block i equals to n_i minus the number of 1-probes in the i-th block. Therefore, we can bound the number of bit vectors compatible with x by

$$\frac{C}{2^U} \le \frac{\binom{u_1}{v_1}}{2^{u_1}} \frac{\binom{u_2}{v_2}}{2^{u_2}} \cdots \frac{\binom{u_p}{v_p}}{2^{u_p}} \tag{2}$$

Let us classify blocks into two categories: determined and undetermined. We call block i *determined* if $u_i \le U/(2p)$ (intuitively, when it has less than half of the "average" number of unprobed bits) and call it *undetermined* otherwise. Let d be the number of determined blocks. Then

$$U = \sum_i u_i \le dU/(2p) + (p-d)k$$

hence

$$d \le p\frac{k - U/p}{k - U/(2p)} \le (1-a)p$$

where $0 < a < 1$ is a constant. Thus, there is at least a constant fraction of undetermined blocks. We bound

$$\frac{\binom{u_i}{s_i}}{2^{u_i}} < 1$$

for determined blocks, and

$$\frac{\binom{u_i}{s_i}}{2^{u_i}} \le \frac{\binom{u_i}{u_i/2}}{2^{u_i}} < \frac{b}{\sqrt{u_i}} \le \frac{b}{\sqrt{U/(2p)}} < \frac{c}{\sqrt{\lg n}}$$

for undetermined blocks using Stirling formula, where $b > 0$ and $c > 0$ are constants. Thus (2) can be bounded by

$$\frac{C}{2^U} \le \left(\frac{c}{\sqrt{\lg n}}\right)^{ap} \tag{3}$$

Recall that both C and U depend on x, so that $U(x) = n + r - \mathrm{depth}(x)$. We can compute the following sum

$$\sum_{x \text{ is a leaf}} 2^{U(x)} = 2^{n+r} \sum_{x \text{ is a leaf}} 2^{-\mathrm{depth}(x)} = 2^{n+r}$$

The total number of bit vectors B compatible with some leaf is at most

$$\sum_{x \text{ is a leaf}} C(x) \le 2^{n+r} \left(\frac{c}{\sqrt{\lg n}}\right)^{ap} \tag{4}$$

However, each bit vector has to be compatible with at least one leaf

$$2^n \le \sum_{x \text{ is a leaf}} C(x) \tag{5}$$

Thus

$$r = \Omega(n \lg \lg n / \lg n)$$

The index presented in the previous section matches this lower bound up to a constant factor.

3.2 Select Index

In this section, we give an argument that techniques from Miltersen [3] that establish a lower bound $O(\lg n)$ for select index when we allow to probe $O(\lg n)$ bit probes

$$r = \Omega(n / \lg n) \tag{6}$$

cannot be improved to obtain

$$r = \Omega(n \lg \lg n / \lg n) \tag{7}$$

Miltersen used only bit vectors that only have $O(n/\lg n)$ 1-bits. However, for such vectors, we can construct an index of size $O(n/\lg n)$ that allows $O(1)$ select queries. Let us divide B into p subregions of size $(\lg n)/2$, for each subregion we count number of 1 bits in it (denote it by n_i) and represent it in unary. We construct the following bit vector

$$L = 1^{n_1}01^{n_2}0 \ldots 01^{n_p}$$

of length $p + O(n/\lg n) = O(n/\lg n)$. To perform $\text{select}_B(j)$ on B, we first find $x = \text{select}_L(i)$ and then $i = x - \text{rank}_L(x)$, the number of 0-bits before the position x in L. Hence i gives us the number of the block of B where j-th 1-bit is located (denote the block K). Next, we compute $z = \text{select0}_L(i)$ (where $\text{select0}_L(j)$ gives position of j-th occurrence of 0-bit in L), the starting position of i-th block in L. And then compute $t = x - z$, so that j-th 1-bit of B is t-th 1-bit of K. Finally, $select_K(j)$ can be done by a lookup to a table of size $\sqrt{n}(\lg \lg n)^2$ bits that stores results of all possible select queries for all possible blocks. Note that rank and select on L requires at most $o(n/\lg n)$ bits in addition to storing L as we discussed in the previous section. Thus, the total space requirement for the index is $O(n/\lg n)$ bits. It follows that for such bit vectors B indexes of size $O(n/\lg n)$ are optimal.

Let us apply a combinatorial technique similar to the one that we used for rank index to show a stronger lower bound. Fix the number of probes to the bit vector B to be $t = f \lg n$ (for some constant $f > 0$) that select algorithm uses and let $k = t + \lg n$ as before.

Let us restrict ourselves to bit vectors B of cardinality $n/2$ ($n/2$ bits are 0 and $n/2$ bits are 1). Let us perform the following $p = n/(2k)$ queries: for each $1 \leq i \leq p$ we simulate $\text{select}(ik)$. Similarly, we construct choices tree for these queries. To compute the number of compatible bit vectors for a given leaf, we split each string B into p blocks, i-th block is from position $\text{select}_B((i-1)k) + 1$ to position $\text{select}_B(ik)$ (we define $\text{select}_B(0) = 0$ for convenience). Note that there are exactly k ones in each block. The number of unprobed bits U for the whole string is at

least $n-pt = n(1-t/k) = \Omega(n)$. We can count the number of compatible nodes C for each leaf x by applying the same technique as for rank, and obtain (similarly to (3))

$$\frac{C}{2^U} \leq \left(\frac{c}{\sqrt{\lg n}}\right)^{ap}$$

where $0 < a < 1$ and $0 < c$ are positive constants. Next, we can obtain the bound on the total number of bit vectors B that are compatible with at least one node in the choices tree. Similarly to (4), we have

$$2^{n+r}\left(\frac{c}{\sqrt{\lg n}}\right)^{ap}$$

The total number of bit vectors we are considering is $\binom{n}{n/2}$, thus

$$\binom{n}{n/2} \leq 2^{n+r}\left(\frac{c}{\sqrt{\lg n}}\right)^{ap}$$

and hence

$$r = \frac{\lg\binom{n}{n/2}}{n}\Omega\left(\frac{n\lg\lg n}{\lg n}\right) = \Omega\left(\frac{n\lg\lg n}{\lg n}\right)$$

We state the results for rank and select indexes as the following

Theorem 1. *Let B be a bit string of length n. Assume that there is an algorithm that uses $O(\lg n)$ bit probes to B (plus unlimited access to an index of size r and unlimited computation power) to answer rank (select) queries. Then $r = \Omega(\frac{n\lg\lg n}{\lg n})$.*

4 Density-Sensitive Lower Bounds

In this section, we consider the case where the bit vector B contains some fixed number m of 1-bits and express lower bounds for rank and select in terms of m. We will use techniques similar to the previous section, however, the calculations are slightly more involved in this case. We will a lower bound for rank and select query and omit the proof for select query.

First, we assume that all nodes in the choices tree are at the same level $pt+r$, i.e. on every root to leaf path the rank algorithm probes exactly pt bits. If some node x is z levels above it, we perform z fake probes, in order to split it into 2^z nodes at the required level, so that $U = n - pt$ for all leaves. We will choose parameter p, such that $pt \leq n/2$, so that at least half of the bits are unprobed at the end. We will partition all the leaves x into m groups depending on the total number of 1-probes on the root to leaf path to x (excluding the first r levels for the index). Let G_y be the group of leaves for which we performed y 1-probes. Clearly, $|G_y| \leq 2^r\binom{pt}{y}$. For each leaf $x \in G_y$ we can bound the number of compatible bit vectors by:

$$\binom{u_1}{v_1}\binom{u_2}{v_2}\cdots\binom{u_p}{v_p} \tag{8}$$

where

$$u_1 + u_2 + \ldots + u_p = U \tag{9}$$

$$v_1 + v_2 + \ldots + v_p = V \tag{10}$$

where $U = n - pt$ is the number of unprobed bits and $V = m - y$. Similar to the previous section, u_i denotes the number of uncovered bits in i-th block and $v_i \leq u_i$ denotes the sum of these bits. Recall that v_i equals to n_i minus number of 1-probes in i-th block, and hence is fixed for a given leaf. We will combine blocks into larger *superblocks* as follows. The 1-st superblock will contain blocks $1, 2, \ldots, z_1$, such that $k \leq u_1 + u_2 + \ldots u_{z_1} \leq 2k$, the i-th superblock will contain blocks $z_{i-1}, \ldots, z_i$ such that $k \leq u_i^s \leq 2k$, where

$$u_i^s = u_{z_{i-1}+1} + u_{z_{i-1}+2} + \ldots + u_{z_i}$$

is the *size* of i-th superblock. Note that this is always possible, since $u_i \leq k$ for all i. Let q be the number of superblocks, clearly $n/(4k) \leq q \leq n/k$ (equivalently $p/4 \leq q \leq p$), since $U \geq n/2$.
For each superblock, we will use the inequality

$$\binom{u_{z_{i-1}+1}}{v_{z_{i-1}+1}}\binom{u_{z_{i-1}+2}}{v_{z_{i-1}+2}} \ldots \binom{u_{z_i}}{v_{z_i}} \leq \binom{u_i^s}{v_i^s}$$

where

$$v_i^s = v_{z_{i-1}+1} + v_{z_{i-1}+2} + \ldots + v_{z_i}$$

So that

$$\binom{u_1}{v_1}\binom{u_2}{v_2} \ldots \binom{u_p}{v_p} \leq \binom{u_1^s}{v_1^s}\binom{u_2^s}{v_2^s} \ldots \binom{u_q^s}{v_q^s} \tag{11}$$

Observe that for any $q_1 < p_1$ and $p_2 < q_2$

$$\frac{\binom{p_1}{q_1+1}\binom{p_2}{q_2}}{\binom{p_1}{q_1}\binom{p_2}{q_2+1}} = \frac{\frac{p_1-q_1}{q_1+1}}{\frac{p_2-q_2}{q_2+1}}$$

That is

$$\binom{p_1}{q_1+1}\binom{p_2}{q_2} > \binom{p_1}{q_1}\binom{p_2}{q_2+1}, \text{ if } \quad \frac{q_1+1}{p_1+1} < \frac{q_2+1}{p_2+1}$$

We can interpret this inequality as follows. Let us maximize the product (11) with fixed values of u_i^s's, subject to the constraint $v_1^s + v_2^s + \ldots + v_q^s = V$. The point $(v_1^s, v_2^s, \ldots, v_q^s)$ is a local maximum if we cannot increase v_i^s by 1 and decrease v_j^s by 1 for some $i \neq j$, so that (11) increases. Intuitively, at a local maximum all fractions v_i^s/u_i^s are roughly equal for otherwise we can transfer 1 from smaller fractions to bigger ones. We can show (proof omitted) that if $V > 2q$ then $|v_i^s/u_i^s - V/U| < 2/k$ (recall that all u_i^s satisfy $k \leq u_i^s \leq 2k$, so that each "transfer" does not change each of v_i^s/u_i^s fractions by more than $1/k$).

We will use the Stirling approximation

$$\binom{u}{v} = \Theta\left(\frac{\sqrt{u}}{\sqrt{v}\sqrt{u-v}}\frac{(u/e)^u}{(v/e)^v((u-v)/e)^{u-v}}\right)$$
$$= \Theta\left(\frac{1}{\sqrt{v}}\left(\frac{1}{\xi}\right)^v\left(\frac{1}{1-\xi}\right)^{u-v}\right)$$

where $\xi = v/u$, here we assumed that v is large enough, so that we can use the Stirling approximation for $v!$, but $v/u \le 1/2$. Now we can bound

$$\frac{\binom{u_1^s}{v_1^s}\binom{u_2^s}{v_2^s}\cdots\binom{u_q^s}{v_q^s}}{\binom{U}{V}} = 2^{\Theta(q)}\left(\frac{1}{kV/U}\right)^{q/2}\left(\frac{\phi}{\psi}\right)^V\left(\frac{1-\phi}{1-\psi}\right)^{U-V} \tag{12}$$

where $\phi = V/U \le 1/2$ and ψ is some number, such that $|\psi - \phi| < 2/k$ (again, we assumed $V/U \le 1/2$). The latter expression is less than

$$2^{\Theta(q)}\left(\frac{1}{kV/U}\right)^{q/2}\left(1+\frac{4}{kV/U}\right)^V\left(1+\frac{4}{k}\right)^U \le$$
$$\le 2^{\Theta(q)}\left(\frac{1}{kV/U}\right)^{q/2}2^{\Theta(U/k)} \le 2^{\Theta(q)}\left(\frac{1}{kV/U}\right)^{q/2}$$

Here we used two facts (i) $(1+1/\alpha)^\beta = 2^{\Theta(\beta/\alpha)}$ for large enough α; and (ii) $U/k \le q/2$.

Note that there are could have at most pt different groups of leaves ($y \le pt$). For a given group, we bound the total number of compatible bit vectors by (12). Denote $V^* = m - pt$, and note that $V = m - y \ge V^*$. So that the total number of compatible bit vectors for all groups is at most

$$2^r\sum_{y=0}^{pt}\binom{pt}{y}\binom{n-pt}{m-y}\left(\frac{O(1)}{kV^*/U}\right)^{\Theta(p)} = 2^r\binom{n}{m}\left(\frac{O(1)}{kV^*/U}\right)^{\Theta(p)}$$

However, all possible bit vectors of length n with m ones have to be compatible with at least one leaf, so that

$$r = \Omega(p\lg\frac{n(m-pt)}{p(n-pt)}) = \Omega(p\lg(m/p-t))$$

Choosing $p = m/(2t)$ gives

$$r = \Omega((m/t)\lg t)$$

Essentially the same technique is applicable to the case of select index (proof omited).

Theorem 2. *Let B be a bit string of length n with m ones in it. Assume that there is an algorithm that uses t bit probes to B (plus unlimited access to an index of size r and unlimited computation power) to answer rank (select) queries. Then $r = \Omega((m/t)\lg t)$.*

Note that this theorem gives an optimal lower bound for the case of constant density bit vectors (i.e. when $m/n = \Theta(1)$).

5 Acknowledgments

We thank Ian Munro, Prabhakar Ragde, and Jeremy Barbay for fruitful discussions and proof reading this paper. We also thank Srinivasa S. Rao for bringing the problem to our attention and pointing out the results of Miltersen.

References

1. David R. Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, 1996.
2. Guy Jacobson. *Succinct Static Data Structures*. PhD thesis, Carnegie Mellon University, January 1989.
3. Peter Bro Miltersen. Lower bounds on the size of selection and rank indexes. In *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 11–12, 2005.
4. J. Ian Munro, Venkatesh Raman, and S. Srinivasa Rao. Space efficient suffix trees. In *Proceedings of the 18th Conference on the Foundations of Software Technology and Theoretical Computer Science*, volume 1530 of *Lecture Notes in Computer Science*, pages 186–196. Springer, 1998.
5. J. Ian Munro, Venkatesh Raman, and S. Srinivasa Rao. Space efficient suffix trees. *Journal of Algorithms*, 39(2):205–222, 2001.
6. Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 233–242, 2002.