New Algorithms for Subset Query, Partial Match, Orthogonal Range Searching and Related Problems

Moses Charikar[1], Piotr Indyk[2], and Rina Panigrahy[3]

[1] Princeton University
moses@cs.princeton.edu
[2] MIT
indyk@theory.lcs.mit.edu
[3] Cisco Systems
rinap@cisco.com

Abstract. We consider the *subset query* problem, defined as follows: given a set $\mathcal{P}$ of N subsets of a universe U, $|U| = m$, build a data structure, which for any *query* set $Q \subset U$ detects if there is any $P \in \mathcal{P}$ such that $Q \subset P$. This is essentially equivalent to the partial match problem and is a fundamental problem in many areas. In this paper we present the first (to our knowledge) algorithms, which achieve non-trivial space and query time bounds for $m = \omega(\log N)$. In particular, we present two algorithms with the following tradeoffs:

- $N \cdot 2^{O(m \log^2 m \sqrt{c/\log N})}$ space, and $O(N/2^c)$ time, for any c
- Nm^c space and $O(mN/c)$ query time, for any $c \leq N$

We extend these results to the more general problem of orthogonal range searching (both exact and approximate versions), approximate orthogonal range intersection and the exact and approximate versions of the nearest neighbor problem in ℓ_∞.

1 Introduction

The *subset query* problem is defined as follows: given a set $\mathcal{P}$ of N subsets of a universe U, $|U| = m$, build a data structure, which for any *query* set $Q \subset U$ detects if there is any $P \in \mathcal{P}$ such that $Q \subset P$. This problem is of fundamental importance in many areas. In information retrieval, it occurs in applications which allow the user to search for documents containing a given set of words (e.g., as in Google). In the field of databases, it corresponds to the *partial match problem*, which involves searching for records which satisfy specified equality constraints[1] (e.g., "Sex=Male and City=Boston"). The partial match problem is also of large interest for IP packet classification, where the goal is to classify a packet depending on its parameters (source and destination address, length etc) [1, 6–8, 16].

[1] For an easy equivalence between the partial match problem and the subset query problem, see Preliminaries.

Due to its high importance, the subset query and partial match problems have been investigated for quite a while. It is easy to see that this problem has two fairly trivial solutions: (a) store all answers to all queries (requiring 2^m space and $O(m)$ query time), or (b) scan the whole database for the answer (requiring linear storage and $O(Nm)$ query time). Unfortunately, both solutions are quite unsatisfactory in most applications. The first non-trivial result for this problem has been obtained by Rivest [14, 15]. He showed that the 2^m space of the trivial "exhaustive storage" solution can be somewhat improved for $m \leq 2 \log N$. He also presented a trie-based algorithm using linear storage, which achieved sublinear query time when the database content is generated at *random*.

Unfortunately, since those early results, there has been essentially no progress on this problem to this date[2]. Consequently, it is believed (e.g., see [5] or [10] and the references therein), that the problem inherently suffers from the "curse of dimensionality" , i.e., that there is no algorithm for this problem which achieves *both* "fast" query time and "small" space. Various variants of the conjecture exist, depending on the definition of "fast" and "small". Recently it has been shown [5] that the problem requires space superpolynomial in n if the query time (in the cell-probe model) is $o(\log m)$. However, there is an exponential gap between the lower bound and the upper bounds.

In sharp contrast to these problems, recent research has yielded fairly good bounds for the approximate nearest neighbor problem in Euclidean spaces, a fundamental problem that is very challenging for points in high dimensions. In particular, [12] and [11] provided the first algorithm for $(1+\epsilon)$-approximate nearest neighbor in l_2^d or l_1^d, with polynomial storage and $(1/\epsilon + d + \log N)^{O(1)}$ query time. For l_∞^d, [9, 10] gave a 3-approximate nearest neighbor algorithm using $N^{O(\log d)}$ storage and having $(d + \log N)^{O(1)}$ query time.

In this paper we present the first (to our knowledge) algorithms, which achieve non-trivial space and query time bounds for $m = \omega(\log N)$. In particular, we present two algorithms with the following tradeoffs:

- $N \cdot 2^{O(m \log^2 m \sqrt{c/\log N})}$ space, and $O(N/2^c)$ time, for any c
- Nm^c space and $O(mN/c)$ query time, for any $c \leq N$

For reporting queries, our algorithms output a collection of pointers to precomputed lists. Thus they do not need to output each individual point. The first algorithm is interesting when the goal is to achieve a $o(N)$ query time. For example, if we set $c = \log N/\mathit{poly}\log m$, then we can achieve an algorithm with space $N2^{m/\mathit{poly}\log m}$ (i.e., subexponential in m), with query time $N^{1-1/\mathit{poly}\log m}$. For $m = O(\log^{1.4} N)$ (and a different value of c), the first algorithm yields a data structure with space $N^{1+o(1)}$ and query time sublinear in N. On the other hand, the second algorithm is interesting when the goal is to achieve a $o(mN)$ query time. This becomes interesting if m is fairly large compared to N, e.g. when $m > N^{(1+\alpha)/2}$ for $\alpha > 0$. In this case, by setting $c = \sqrt{N}$ we get significantly subexponential space requirements (i.e., $2^{O(m^{1/(1+\alpha)} \log m)}$), while achieving significantly sublinear query time of $O(m\sqrt{N})$.

[2] From the theoretical perspective - many data structures has been designed which improve the space or the query time in practice.

We also mention that the first result shows that the strongest version of the curse-of-dimensionality conjecture (which interprets $2^{o(m)}$ space as "small" and $o(N)$ query time as "fast", with $m = \omega(\log N)$) is *false.*

We extend the above results to a more general problem of *orthogonal range searching* [2], which involves searching for points contained in a d-dimensional query rectangle. Range searching is one of the most investigated problems in computational geometry. It is known [13] that one can construct a (range-tree) data structure of size $O(N \log^d N)$ which answers range queries in time $O(\log^{d-1} N)$; sub-logarithmic improvements to these bounds are also known.

We show that the second of the aforementioned subset query algorithms can be extended to solve orthogonal range search within the same bounds (with m replaced by d). This yields an algorithm with Nd^c space and $O(dN/c)$ query time. The algorithm can be immediately used to solve the nearest neighbor problem in l_∞^d with space bound unchanged and query time multiplied by $\log n$. Moreover, a special case of orthogonal range search with *aligned* query rectangles (i.e. whose projection on each dimension is of the form $[x2^i, (x+1)2^i])$ reduces directly to the partial match problem with $m = s$, where s is the number of bits required to specify each point in the database. This result is interesting for range queries in databases where queries require exact matches on most fields and specify ranges in a few fields. In addition, we show that $(1+\epsilon)$-approximate orthogonal range searching can be reduced to the partial match problem with $m = O(s/\epsilon)$. This reduction gives interesting trade-offs when combined with the first of the aforementioned subset query algorithms. The results also apply to the $(1+\epsilon)$-approximate nearest neighbor problem in l_∞. We can also show that a $(1+\epsilon)$-approximate range intersection can be reduced to a partial match problem with $m = \tilde{O}(s/\epsilon^4)$.[3]

In addition, we show that orthogonal range searching in dimension d can be reduced to the subset query problem with $m = O(d^2 \log^2 N)$. This demonstrates the close relationship between the two problems. Moreover, combined with the result of [9] (who showed that the subset query problem can be reduced to c-approximate nearest neighbor in l_∞^d for $c < 3$) it implies an interesting fact: if we efficiently solve the c-approximate nearest neighbor in l_∞^d for $c < 3$, then we can solve this problem *exactly* with similar resources.

2 Preliminaries

In this section we show that several superficially different problems are essentially equivalent to the subset query problem. These easy equivalences are widely known. In this paper we switch between these problems whenever convenient.

Partial match problem. Given a set $\mathcal{D}$ of N vectors in $\{0,1\}$, build a data structure, which for any *query* $q \in \{0,1,*\}$, detects if there is any $p \in \mathcal{D}$ such that q matches p. The symbol "*" acts as a "don't care" symbol, i.e., it matches

[3] We use $\tilde{O}(n)$ to denote $O(n\, poly \log n)$.

both 0 and 1. Note that one can extend any algorithm solving this problem to handle non-binary symbols in vectors and queries, by replacing them with their binary representations.

It is easy to reduce partial match problem to the subset query problem. To this end, we replace each $p \in \mathcal{D}$ by a set of all pairs (i, p_i), for all $i = 1 \ldots m$. In addition, we replace each query q by a set of all pairs (i, q_i) such that $q_i \neq *$. The correctness is immediate.

It is also see how to reduce subset query to the partial match problem. This is done by replacing each database set P by its characteristic vector, and replacing query set Q by its characteristic vector in which 0's are replaced with $*$'s.

Containment query problem. This problem is almost the same as the subset query problem, except that we seek a set P such that $P \subset Q$. Since P is a subset of Q iff its complement is a superset of the complement of Q, the subset and the containment query problems are equivalent as well.

3 Algorithm for Set Containment

In this section, we present an algorithm for set containment. A naive algorithm to answer set containment queries on a universe of size m is to write the query set as a bit vector and index it into an array of size 2^m. Here each array location points to a list L of sets from the database contained in the set corresponding to the array location.

To reduce the space complexity one can use random sampling to hash the query sets into a smaller hash table by focusing on a random subset of the universe. Say we random sample with probability p and obtain pm elements of the universe. One can now construct an array of size 2^{pm} based only on these sampled elements. Again, the query set maps to an array location which contains a list L of database sets. However, we can only guarantee that a set in list L is contained in Q with respect to the sampled elements. In fact, one can show that the sets in L are *almost* contained in Q. In other words the difference from Q of a set in L is likely to be small. (A set with a large difference from Q is likely to be eliminated since the random sample would contain an element from the difference.) The fact that sets in L have small differences from Q implies a bound on the number of distinct differences from Q. We choose our sampling probability such that the number of distinct differences from Q of sets in L is small (sublinear in N). However this property by itself does not help in yielding a query time sublinear in N. This is because even though the number of possible differences from Q is small, the number of sets in L may be quite large as there could be several sets that have the same difference from Q.

However, the fact that the sets in L have small differences from Q suggests that the sets in L could be grouped according to their difference from Q. It is not clear how to do this since the differences depend on the query set Q and a number of query sets Q map onto the same location in the array. However, this idea can be adapted to yield an efficient data structure. For the list L we construct a baseline set and work with differences from that set. This baseline set R which we call a *representative set* is such that sets in L have small differences from R. We group the sets in L by their differences from R. Now assuming a query set Q contains R we only need to look at which of the differences from R are contained in Q.

However, we cannot ensure that every query set Q that hashes to a list L contains its representative set R. It turns out that we can relax this condition to say that Q almost contains R, i.e. the difference $R - Q$ is small.

We now build a second level hash table to organize the collection L. Entries in this second level table are indexed by small subsets of R. The query set Q maps onto the location corresponding to $R - Q$ (note that this is a small subset by the properties of the representative set). This location is associated with the subcollection $L' \subseteq L$ of sets that do not intersect $R - Q$. Further, the sets in L' are grouped according to their difference from R. Since the sets in the original collection L had small differences from R, the sets in the subcollections L' also have small differences. For each group of sets with the same difference from R, we check to see if the difference from R is completely contained in Q. If so, all the sets in the group are contained in Q; if not, all sets are not contained in Q. This is depicted in Figure 1. The query time is determined by the number of groups (i.e. number of differences from R) that must be examined by the algorithm. The number of differences can be bounded by the fact the sets have small differences from R; we choose our parameters so that this is sublinear in N.

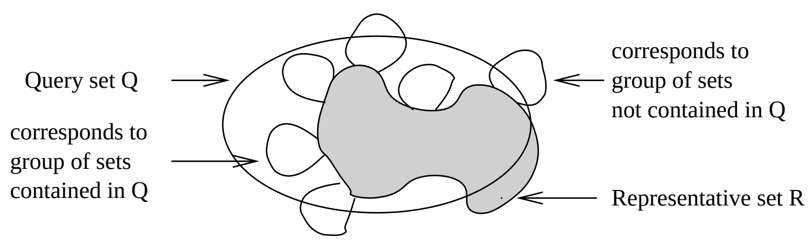

Fig. 1. *The second level hash table in the containment query data structure. The "ears" around R represent collections of sets that have the same difference from R*

3.1 Technical Details

We start by defining relaxed notions of containment and disjointness:

Definition 1 (Almost contains). *A set Q is said to* x-almost contain *set P if $|P - Q| \leq x$.*

Definition 2 (Almost disjoint). *A set Q is said to be* x-almost disjoint *from set P if $|P \cap Q| \leq x$.*

We will use parameters p and x in presenting the algorithm details; values for these parameters will be picked later. For brevity, we will use *almost contains* and *almost disjoint* to mean x-almost contains and x-almost disjoint respectively.

We first select a subset $S \subseteq U$ by picking $p \cdot m$ elements uniformly and at random from U. The first level hash table is indexed by subsets of S. Query set Q is mapped to the location indexed by $Q \cap S$. The size of this table is 2^{pm}. The table location corresponding to subset $Q' \subseteq S$ has associated with it a list L of sets such that $\forall\ P \in L, P \cap S \subseteq Q'$. (These are sets that seem to be contained in Q with respect to the sampled elements.) Note that $P \in L \Leftrightarrow S \cap (P - Q) = \emptyset$.

If a set P is not almost contained in Q (i.e. $|P - Q| > x$) then $\mathbf{Pr}[P \in L] \leq 2^{-px}$. The expected number of sets in L that are not x-almost contained in Q is at most $N \cdot 2^{-px}$. So with probability at least $1/2$, there are at most $2N \cdot 2^{-px}$ sets in the list that are not almost contained in Q (i.e. very few sets have large differences from Q). We will say that a query set Q *is good for* a list of sets L from the database if at most $f = 2N \cdot 2^{-px}$ sets in L are not almost contained in Q. By the preceding discussion, if the query set Q maps onto an array location associated with a list L, with probability at least $1/2$, Q *is good for* L. For the remainder of the discussion, we will assume that this is indeed the case and show how this can be ensured in the end.

Recall that we associate with each list L, a representative set R so that sets in L can be examined efficiently. We now describe the precise properties that R satisfies and an algorithm to construct such a representative set. A set R is said to be *a representative set* for a list L if

1. For any query set Q that is good for L, $|R - Q| \leq g = p \cdot m$.
2. Except for at most $(2/p) \cdot N \cdot 2^{-px}$ sets in L, the others differ from R in at most x/p elements.

Note that the total number of distinct differences of sets in L from R is at most $(2/p) \cdot N \cdot 2^{-px} + \binom{m}{x/p}$. We use the following algorithm to produce such a representative set for a list L.

Construct-Representative(L)

1. $R = \cup_{P \in L} P$.
2. Check if there exists a subset $G \subseteq R, |G| = g$ that is almost disjoint from all sets in L, except for at most f sets.
3. If no such set G exists, stop and output R, else $R \leftarrow R - G$.
4. Go back to step 2.

Claim. Construct-Representative-Set(L) produces a valid representative set for L.

Proof. Consider the final value of R produced by the algorithm. We claim that for all query sets Q that are good for L, $|R - Q| \leq g$. Suppose in fact, there is a query set Q that is good for L such that $|R - Q| > g$. Since Q is good for L, except for at most f sets, the sets in L are almost contained in Q (i.e. have at most x elements outside Q). Let G be any g element subset of $R - Q$. Except for at most f sets in L, the sets in L must be almost disjoint from G. (i.e. have intersection of size at most x). But the algorithm would have detected this subset in Step 2, and would not have terminated in Step 3. By contradiction, R must satisfy the first condition required of a representative set.

Now we establish the second condition. Initially, all sets in L are completely contained in R. For a given set in L, the elements outside the final set R are precisely those elements which are removed from R as a result of discarding the set G in each iteration. The number of iterations is at most m/g since the initial set R has at most m elements and g elements are removed in each iteration. In each iteration at most f sets in L are not almost disjoint from G. This gives a total of $(m/g) \cdot f \leq (2/p) \cdot N \cdot 2^{-px}$ such sets. For the remaining sets, the intersection with the set G in each iteration has size at most x. Thus, in removing G, at most x elements are removed from such a set. So the total number of elements from such a set outside the final R is at most $(m/g) \cdot x \leq x/p$. This proves that R satisfies the second condition required of a representative set.

Recall that to check whether Q contains a set P, we check that **(1)** Q contains P with respect to the elements in R, and **(2)** Q contains P with respect to the elements in $\bar{R}$. To perform the first check, we need to eliminate sets that contains any elements in $R - Q$. Note that $|R - Q| \leq g$ by the properties of the representative set. To facilitate the first check, we keep a second level array with each list L indexed by all possible subsets of R of size at most g. The size of this array is at most $\binom{m}{g} \leq 2^{pm \log m}$. Query set Q indexes into the array location indexed by $R - Q$. This second level array indexes into a sublist of L that contains sets in L disjoint from $R - Q$. To perform the second check, this sublist is grouped by the differences from R, i.e. sets with the same difference from R are grouped together.

The query time for this algorithm is determined by the number of distinct differences from R that the algorithm must examine. This number is at most $(2/p) \cdot N/2^{px} + \binom{m}{x/p}$. Further, the space requirement is $2^{pm} \cdot 2^{pm \log m} \cdot N$. Setting $px = t$ where $t < (\log N)/2$, and solving for $\binom{m}{x/p} = N/2^t$, we get $p = \sqrt{t \log m / \log N}$ The number of distinct differences is now at most $(2/p) \cdot N/2^{px} + \binom{m}{x/p} = 2m \cdot N/2^t + N/2^t = (2m+1)N/2^t$. We also know that with probability $1/2$, Q is *good for* L. So if we have $O(m)$ such arrays every set Q will be good for some list L in one of the arrays with very high probability. Setting $t = 2c \log m$, this gives us a query time of $O(N/2^c)$ and space of $N \cdot 2^{O(m \log^2 m \sqrt{c/\log N})}$. It can be verified that the time to construct the data structure is $N \cdot 2^{O(m \log^2 m \sqrt{c/\log N})}$.

We can obtain better space bounds when the size of the query set is bounded by q. In this case, we can get a query time of $O(N/2^c)$ using space $N \cdot 2^{O(q \log^2 m \sqrt{c/\log N})}$). The basic idea is to replace the random sampling step by a certain construct called a *Bloom filter* [4], used for succinct representation of sets to support membership queries. The details will appear in the full paper.

4 Partial match and orthogonal range search in very high dimensions

In this section we describe the algorithm which achieves $Nm^{O(c)}$ space and $O(mN/c)$ query time, for any $c \leq N$. We first describe in the the context of the subset query problem, and then modify it to work for general orthogonal range search.

The main part of the data structure is a "tree of subsets" of $\mathcal{P}$ (call it T). The root of T is formed by the set $\mathcal{P}$, and each internal node corresponds to a subset of $\mathcal{P}$. We show first how to construct the children of the root. For any $i \in U$, we say that the *attribute* i is *selective* for $\mathcal{P}' \subset \mathcal{P}$, if the number of sets in the family which do not contain i is at least N/c. Let I be the set of attributes which are selective for $\mathcal{P}$. For each $i \in I$ we form a child $\mathcal{P}_i$ of $\mathcal{P}$ by removing from $\mathcal{P}$ all sets which do not contain i. We then apply the same procedure to the new nodes. The procedure stops when there are no further nodes to expand.

Given a query Q, the search proceeds as follows. We start from the root and then proceed further down the tree. At any node (corresponding to a set $\mathcal{P}'$), we check if there is any attribute $i \in Q$ which is selective for $\mathcal{P}'$. If so, we move to $\mathcal{P}'$. Otherwise, we enumerate all pairs (i, j), such that $i \in Q$ and $P_j \in \mathcal{P}'$ does not contain i. Note that we can enumerate all such pair in time linear in their number. For every such pair, we mark the set P_j as *not* an answer to the query Q. At the end, we verify if the number of marked nodes in less than N; if so, we output YES, otherwise we output NO.

The complexity bounds for the above data structure can obtained as follows. Firstly, observe that any parent-child pair $\mathcal{P}', \mathcal{P}''$ we have $|\mathcal{P}''| \leq \mathcal{P}'| - N/c$. Thus, the depth of the tree T is at most c, and thus its size is bounded by m^c (in fact, slightly better bound of $\binom{m}{c}$ can be shown). Moreover, the tree can be traversed from the root to any (final) node in $O(m + N/c)$ time. This is due to the fact that the selectivity property is monotone, and thus once an attribute $i \in Q$ ceases to be selective, we never have to check it again. Finally, the cost of enumerating the pairs (i, j) is at most mN/c, since the numbers of j's per each $i \in Q$ is at most N/c. The bounds follow.

The data structure can be generalized to orthogonal range queries in the following way. Firstly, observe that by increasing the query time by an additive factor of $O(d \log n)$ we can assume that all coordinates of all database points are distinct numbers from $\{1 \dots nd\}$ (this is based on folklore technique of replacing each number by its rank in a sorted order). We create a tree T in a similar way as before. Moreover, it is sufficient to consider query rectangles which are products of left-infinite intervals $[-\infty, a_i]$ for $i = 1 \dots d$. Starting from the root node, for each dimension i, we find a "threshold" t_i such that the number of points with the i-th coordinate greater than t_i is exactly N/c. Then we construct the children sets $\mathcal{P}_i$ by removing from $\mathcal{P}$ all points with the i-th coordinate greater than t_i, and apply the same procedure recursively to the newly created nodes.

In order to answer a query $\prod_i [-\infty, a_i]$, we proceed top-down as before. At any node $\mathcal{P}'$ we check if there is any i such that $a_i < t_i$. If so, can move to the node $\mathcal{P}'_i$. Otherwise, we have $a_i > t_i$ for all i's (we can assume there are not ties). In this case we enumerate all pairs (i, j) such that the i-th coordinate of the j-th point is greater than t_i. Among such pairs, we choose those for which the i-th coordinate is greater than a_i, and mark the j-th point as not-an-answer.

The analysis of space and time requirements is as for the first algorithm.

5 Approximate Orthogonal Range Search

The special case of orthogonal range search where the coordinates in each dimension can only take on values $\{0, 1\}$ is exactly the partial match problem. In this case, points in a d dimensional space can be represented by bit vectors of length d and a query rectangle can be represented as a ternary vector with don't cares or '*' in certain dimensions. As explained in Section 2, the partial match problem can be reduced to set containment. The general case of d dimensional orthogonal range search also reduces to the partial match problem when the query rectangles are well aligned (i.e. the projection on each dimension is of the form $[i2^j, (i + 1)2^j]$.) In this case too, points can be represented as bit vectors and a query rectangle can be represented as a ternary vector (with coordinates in $\{0, 1, *\}$), creating an instance of the partial match problem. This gives us an algorithm for high dimensional orthogonal range search where the query rectangle is well aligned. Specifically if s bits are required to represent a point in the space then an aligned orthogonal range search can be performed in sublinear time by using $2^{(s \log^2 s/\sqrt{\log N})}$ space. Note that most algorithms for range search use a tree based recursive space partitioning scheme whereas our algorithm is based on hashing.

We now extend this result to a $(1+\epsilon)$-approximation algorithm for a general d dimensional orthogonal range search problem where the query rectangle may not be aligned. By a $(1 + \epsilon)$-approximation we mean that we will approximate the query rectangle by larger rectangle containing the query rectangle where each side is blown up by a factor of at most $(1 + \epsilon)$.

Theorem 1. *A $(1+\epsilon)$-approximation of an instance of orthogonal range search can be reduced to an instance of set containment with $m = O(s/\epsilon)$ elements in the universe where s is the number of bits in a binary representation of a single point in the high dimensional space.*

To achieve this we first generalize the partial match problem from binary vectors to k-ary vectors. The query now consists of a range (subinterval) in $[0..k - 1]$ in each dimension. The database now consists of d dimensional k-ary vectors. A vector is set to match the query if it satisfies the range specified in the query for every dimension. We can reduce this problem to set containment by using k elements per dimension, one for each possible value. We construct the query set by looking at the range specified in each dimension and including only the elements that fall in the range. As for a point in the database for each dimension we include the element corresponding to the value for that dimension. This reduction gives a total of $m = k \cdot d$ elements.

Now look at the range specified in a query rectangle on any one dimension. This range consists of a lower bound, l and an upper bound, u. Express these bounds in k-ary representation (base k). Look at the most significant digit where they differ. Let b_l and b_u be the values of that digit in the two bounds, l and u. This gives us a subinterval $[b_l, b_u]$ contained in $[0..k - 1]$. If the length of this subinterval is large say at least $k/2$ then by ignoring the later digits we get a

$4/k$ approximation to the range specified in the query (we get an error of one on each side). If this condition holds for all dimensions we get a $2/k$ approximation by reducing it to the k-ary partial match problem.

However this condition may not be true in every dimension. We force this condition by having multiple representations of every coordinate. Say p bits are required to represent a coordinate in binary. By grouping these p bits into blocks of size $\log k$ we get a k-ary representation (assume k is a power of 2 and is >4). Now look at a range $[l, u]$ of length $r = u - l$ specified on this coordinate. Let the ith digit be the most significant digit of the k-ary representation of r. We want this digit to be large. Further we want the ith digit of l to be small so that by adding r to l we do not affect the previous digit. So we will insist that the ith digit of l is at most $k/2$ and the ith digit of r is between $k/8$ and $k/4 - 1$. The first condition can be achieved by having two representations of l, that is l and $l + D$ where $D = k/2.k/2 \ldots k/2$ repeated $p/\log k$ times (in k-ary representation). Clearly either l or $l + D$ has its ith digit at most $k/2$. We can ensure that the ith digit of r is large by having $\log k$ different representations; that is shift the groupings of bits into k-ary digits by $1, 2, \ldots \log k$ bits. In at least one of these representations the most significant digit of r is at least $k/8$ and $< k/4$. So we have a total of $2 \log k$ representation obtained by the different shifts and the displacement. In at least one of these by ignoring all digits after the most significant digit of r we get a $16/k$ approximation to the length of the range. We concatenate these $2 \log k$ different representations of the range $[l, u]$. For representations that do not satisfy the above conditions we place the don't care range, $[0..k - 1]$, for all its digits. We thus get a k-ary vector consisting of $(p/\log k) \cdot 2 \log k = 2p$ digits for each coordinate. Similarly each coordinate value of any point in database requires $2p$ k-ary digits. So if a d dimensional point requires s bits to specify it, we will get $2s$ k-ary digits after concatenating the $2 \log k$ different representations for each coordinate. After reducing this to set containment we get $m = 2s \cdot k$ elements. The amount of space required to answer this in sub linear time is $2^{2s \cdot k \cdot poly \log(sk)/\sqrt{\log N}}$. The approximation factor $\epsilon = 16/k$. The space required in terms of ϵ is $2^{\tilde{O}(s/\epsilon)/\sqrt{\log N}}$.

Alternatively one could have $(2 \log k)^d$ sets of data structures and after trying out the $2 \log k$ different representations for each coordinate use the appropriate data structure. Each data structure would now correspond to a universe of size $m = 2sk/\log k$. In the full paper, we will discuss applications to the approximate nearest neighbor problem in ℓ_∞ and explain how the ideas for orthogonal range search can be extended to orthogonal range intersection and containment.

6 Reduction from orthogonal range search to the subset query problem

Let $G = (X, E)$ and $G' = (X', E')$ be a directed graph, $|X| \leq |X'|$. We say that $f, g : X \to X'$ embed G into G', if for all $p, q \in X$ we have $(p, q) \in E$ iff $(f(p), g(q)) \in E'$; this definition can be extended to randomized embeddings in a natural way. Let $G = (X, \subset)$ be a poset defined over the set X of all intervals $I \subset \{1 \dots u\}$, and let $G' = (X', \mathcal{E})$ be a directed graph over the set $X' = P_s(\{1 \dots v\})$ ($P_s(Z)$ here denotes all s-subsets of Z) such that for any $A, B \in X'$ we have $(A, B) \in \mathcal{E}$ iff $A \cap B = \emptyset$.

Lemma 1. *There is an embedding of G into G' such that $s = O(\log u)$ and $v = O(u)$.*

Proof. For simplicity we assume u is a power of 2. We start from solving the following problem: design mappings $h_1 : \{1 \dots u\} \to P_s(\{1 \dots O(u)\})$ and $h_2 : X \to P_s(\{1 \dots O(u)\})$ such that $i \in I$ iff $h_1(i) \cap h_2(I) \neq \emptyset$ (intuitively, I will play the role of a "forbidden interval"). To this end, consider a binary tree T with the set of leafs corresponding to the elements of $\{1 \dots u\}$ and the set of internal nodes corresponding to the dyadic intervals over $\{1 \dots u\}$ (i.e., we have nodes $\{1, 2\}$, $\{3, 4\}$ etc. on the second level, $\{1 \dots 4\}$, $\{5 \dots 8\}$ etc on the third level and so on). We define $h_1(i)$ to be the set of all nodes on the path from the root to the leaf i. We also define $h_2(I)$ to be the set of nodes in T corresponding to intervals from the dyadic decomposition of I (i.e., the unique minimum cardinality partitioning of I into dyadic intervals). Note that $|T| = O(u)$.

The correctness of h_1 and h_2 follows from the fact that $i \in I$ iff there is a node on the path from the root to i corresponding to a dyadic interval contained in I.

Now, we observe that $\{a \dots b\} \not\subset \{c \dots d\}$ iff $a \in \{1 \dots c-1\}$ or $a \in \{d+1 \dots u\}$ or $b \in \{1 \dots c-1\}$ or $b \in \{d+1 \dots u\}$. Therefore, we can define $f(\{a \dots b\}) = h_1(a) \cup h_1(b)$ and $g(\{c \dots d\}) = h_2(\{1 \dots c-1\}) \cup h_2(\{d+1 \dots u\})$.

Lemma 2. *There is a randomized embedding of G' into $(P_s(\{1 \dots s^2/\delta\}, \mathcal{E})$ which preserves* non-intersection *between any two elements of G' with probability $1 - \delta$, for $0 < \delta < 1$. The* intersection *is always preserved.*

Proof. It is sufficient to use a random function $h : \{1 \dots v\} \to \{1 \dots s^2/\delta\}$ and extend it to sets.

Consider now the orthogonal range query problem in d-dimensional space. By using the above lemmas we can reduce this problem to the subset query problem for N subsets of $\{1 \dots d^2 \log^2 |P|/\delta\}$ in the following way. For $i = 1 \dots d$ use Lemma 1 to obtain functions f_i and g_i. For simplicity we assume that the values of f_i and f_j do not intersect for $i \neq j$; we make the same assumption for g_i's. For any point $p = (x_1 \dots x_d)$ define $L(p) = \cup_i f_i(\{x_i, x_i\})$. Also, for any box $B = \{a_1 \dots b_1\} \times \{a_2 \dots b_2\} \times \dots \times \{a_d \dots b_d\}$ define $R(B) = \cup_i g_i(\{a_i \dots b_i\})$. We have that $p \in B$ iff $L(p) \cap R(B) = \emptyset$. Since $L(p)$ and $R(B)$ are of size $O(d \log u)$, we can apply Lemma 2 to reduce the universe size to $O(d^2 \log^2 u)$ (say via function h). The condition $A \cap B = \emptyset$ is equivalent to $A \subset \overline{B}$, where $\overline{B}$ is the complement of B. Therefore, with a constant probability, there exists $p \in P$ which belongs to B iff $h(R(B)) \subset \overline{h(L(p))}$. Thus, we proved the following theorem.

Theorem 2. *If there is a data structure which solves the subset query problem for n subsets of $\{1 \dots k\}$ having query time $Q(n, k)$ and using space $S(n, k)$,* *then there exists a randomized algorithm for solving the orthogonal range query problem for n points in $\{1 \dots u\}^d$ with query time $Q(n, O(d^2 \log^2 u))$ and using space $S(n, O(d^2 \log^2 u))$.*

Note that one can always reduce u to $O(n)$ by sorting all coordinates first and replacing them by their ranks.

7 Acknowledgements

Moses Charikar gratefully acknowledges support from NSF ITR grant CCR-0205594 and DOE Early Career Principal Investigator award DE-FG02-02ER25540.

References

1. H. Adiseshu, S. Suri, and G. Parulkar. Packet Filter Management for Layer 4 Switching. *Proceedings of IEEE INFOCOM*, 1999.
2. P. Agarwal and J. Erickson. Geometric range searching and it's relatives. *Advances in Discrete and Computational Geometry*, B. Chazelle, J. Goodman, and R. Pollack, eds., Contemporary Mathematics 223, AMS Press, pp. 1-56, 1999.
3. S. Arya and D. Mount. Approximate Range Searching *Proceedings of 11th Annual ACM Symposium on Computational Geometry*, pp. 172-181, 1995.
4. B. Bloom. Space/time tradeoffs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.
5. A. Borodin, R. Ostrovsky, and Y. Rabani. Lower bounds for high dimensional nearest neighbor search and related problems. *Proceedings of the Symposium on Theory of Computing*, 1999.
6. D. Eppstein and S. Muthukrishnan. Internet Packet Filter Management and Rectangle Geometry. *Proceedings of 12th ACM-SIAM Symp. Discrete Algorithms* 2001, pp. 827-835.
7. A. Feldmann and S. Muthukrishnan, Tradeoffs for Packet classification. *Proceedings of IEEE INFOCOM*, 3:1193–1202. IEEE, March 2000.
8. P. Gupta and N. McKeown. Algorithms for Packet Classification. *IEEE Network Special Issue*, March/April 2001, 15(2):24–32.
9. P. Indyk. On approximate nearest neighbors in non-euclidean spaces. *Proceedings of the 39th IEEE Symposium on Foundations of Computer Science*, pp. 148-155, 1998.
10. P. Indyk. High-dimensional computational geometry. *Ph.D. thesis, Stanford University*, 2001.
11. P. Indyk and R. Motwani. Approximate nearest neighbor: towards removing the curse of dimensionality. *Proceedings of the 30th ACM Symposium on Theory of Computing*, pp. 604–613, 1998.
12. E. Kushilevitz, R. Ostrovsky, and Y. Rabani. Efficient search for approximate nearest neighbor in high dimensional spaces. *Proceedings of the Thirtieth ACM Symposium on Theory of Computing*, pages 614–623, 1998.
13. G. S. Lueker. A data structure for orthogonal range queries. *Proceedings of the Symposium on Foundations of Computer Science*, 1978.
14. R. L. Rivest. *Analysis of Associative Retrieval Algorithms*. Ph.D. thesis, Stanford University, 1974.

15. R. L. Rivest. *Partial match retrieval algorithms*. SIAM Journal on Computing 5 (1976), pp. 19-50.
16. V. Srinivasan, S. Suri, and G. Varghese. Packet classification using tuple space search. *ACM Computer Communication Review* 1999. ACM SIGCOMM'99, Sept. 1999.