Nearest Neighbor Search with Keywords: Compression

Yufei Tao

KAIST

June 3, 2013

In this lecture, we will continue our discussion on:

Problem (Nearest Neighbor Search with Keywords)

Let P be a set of points in $\mathbb{N}^2$, where $\mathbb{N}$ represents the set of integers. Each point $p \in P$ is associated with a set W_p of terms. Given:

- a point $q \in \mathbb{N}^2$,
- an integer k,
- a real value r,
- a set W_q of terms

a k nearest neighbor with keywords (kNNwK) query returns the k points in $P_q(r)$ with the smallest Euclidean distances to q, where

$$P_q(r) \quad = \quad \{p \in P \mid W_q \subseteq W_p \text{ and } dist(p, q) \leq r\}.$$

where $dist(p, q)$ is the Euclidean distance between p and q.

In the previous lecture, we have learned the basic ideas:

- For every term t, make an inverted list $list(t)$ collecting all the points $p \in P$ such that $t \in W_p$.
- Index each $list(t)$ with an R-tree.
- Answer a query via distance browsing.

In this lecture, we will see how to implement these ideas efficiently. In particular, we will discuss:

- How to compress each $list(t)$?
- How to build an R-tree on each $list(t)$.

After resolving the above issues, we will obtain a structure called the spatial inverted index. For simplicity, we will

- focus on $k = 1$, namely, the 1NNwK problem. Extensions to arbitrary k are straightforward and left to you.
- assume that every x- and y-coordinate is in the range of $[0, U - 1]$, where U is a power of 2 that equals the lengths of the x- and y-dimensions.

Each entry in $list(t)$ has the form (id, x, y).

Think

Why do we need id?

Instead, we will store (pid, z) where:

- pid is an integer called a pseudo id.
- z is an integer called a z-value, from which we can obtain x and y uniquely.

Z-order

An encoding that converts a 2d point $p = (x, y)$ to a one dimensional value $z(p)$ called the z-value of p.

Suppose that $x = a_1 a_2 ... a_{\ell-1}$ and $y = b_1 b_2 ... b_{\ell-1}$ in binary, where $\ell = \log_2 U$ (recall that the x- and y- dimensions have domain $[0, U-1]$, and U is a power of 2).

Then, $z(p) = a_1 b_1 a_2 b_2 ... a_{\ell-1} b_{\ell-1}$ in binary.

Example

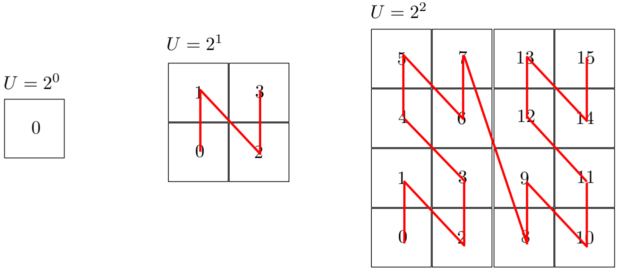
Suppose $x = 13$, $y = 25$, and $U = 32$. Then:

- $x = 01101$ in binary.
- $y = 11001$ in binary.
- $z(p) = 0111100011$ in binary $= 483$ in decimal.

Given $z(p)$, we can decode x and y in a straightforward manner.

Z-order

Pictorial illustrations:



Z-order

The z-order encoding is a form of spatial filling curve, which aims at converting 2d points to 1d values in a proximity preserving manner. This means:

- If two points p_1 and p_2 are close in 2d space, then often $z(p_1)$ and $z(p_2)$ are close.
- If $z(p_1)$ and $z(p_2)$ are close, then often p_1 and p_2 are close in 2d space.

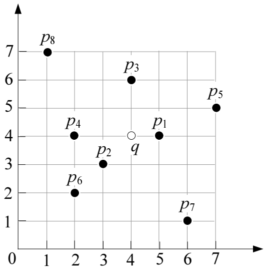
Think

Can you observe from the previous slide that the above sometimes are not true? Do you think that there would exist a "perfect" spatial filling curve that will make the above always true?

Now, let us get back to the 1NNwK problem.

Let us sort all the points in P by z-value. Assign each point $p \in P$ a pseudo id $pid(p)$ that equals the rank of p in the sorted list (i.e., the first point has rank 1, the second has rank 2, ...).

Example: The figure on the right shows the z-values of the data points. Hence, $pid(p_6) = 1$, $pid(p_2) = 2$, ...



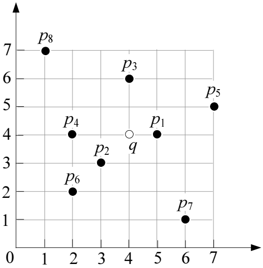
p_6	p_2	p_8	p_4	p_7	p_1	p_3	p_5
12	15	23	24	41	50	52	59

We are ready to explain how to compress an inverted list $list(p)$.

Suppose that S is the set of points in $list(p)$. We sort these points in ascending order of their pseudo ids (and hence, also in ascending order of their z-values). Each entry of the list has the form (pid, z). Apply the "gapping technique" discussed previously, namely, for the i-th pair ($i \geq 2$), store (Δ_{pid}, Δ_z), where Δ_{pid} is the difference between the pid of the pair, and that of the $(i-1)$-th pair, and similarly, for Δ_z.

Store all integers using Elias' Gamma code.

Example:



p	W_p
p_1	$\{a, b\}$
p_2	$\{b, d\}$
p_3	$\{d\}$
p_4	$\{a, e\}$
p_5	$\{c, e\}$
p_6	$\{c, d, e\}$
p_7	$\{b, e\}$
p_8	$\{c, d\}$

p_6	p_2	p_8	p_4	p_7	p_1	p_3	p_5
12	15	23	24	41	50	52	59

Then $list(d)$ has pairs: $(0, 12), (1, 15), (2, 23), (6, 52)$. Hence, we store $(0, 12), (1, 3), (1, 8), (4, 29)$.

Lemma

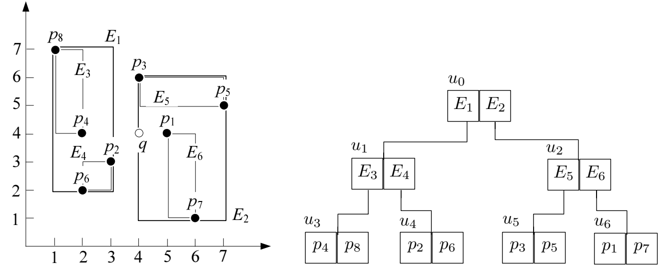
Let $n = |P|$. If $list(p)$ has r points, our compression scheme uses $O(r(\log \frac{n}{r} + \log \frac{U^2}{r}))$ bits.

Next, we describe a simple way to create effective R-trees. Let S be a set of 2d points. Suppose that we have sorted them by z-value in ascending order; let L be the sorted list.

Given a parameter b, let us cut L into blocks of size b, where a block is a subsequence of points in L.

Treat each block as a leaf node. Once all the leaf nodes have been decided, so are the internal nodes.

Example:



p_6	p_2	p_8	p_4	p_7	p_1	p_3	p_5
12	15	23	24	41	50	52	59

We apply this idea to create R-trees on the inverted lists. There is only one issue left. Currently, each inverted list has been compressed using the gapping technique. As a result, if we want to decompress a point p in an inverted list, we must read the bits of all the points before p in the list.

This creates a problem because, in answering a query by distance browsing, we must be able to decompress all the points in a leaf node quickly.

To avoid this problem, we can instead apply the gapping idea locally in each leaf node. Namely, if a leaf node contains a sequence L of points (sorted by z-order), the (pid, z) pair of the first point in L is represented in its original form.

One can show that the extra space overhead thus introduced is limited as long as b is not too small.