Pseudocodes

1 SIX-Regions

Algorithm 1: SIX-Filtering

Divide the space around q in 6 equally sized regions;
initialize $d_i^k \leftarrow \infty$ for each region P_i;
Insert root of facility R*-tree in a min-heap h;
while h is not empty **do**
    deheap an entry e;
    if not prunedByRegion(e) **then** `// see Algorithm 2`
        if e is an intermediate node or leaf **then**
            insert every child c of e in h with key $mindist(q, c)$;
        else
            updateKthNN(e)`// Update` d^k `of the relevant region (see Algorithm 3);`

Algorithm 2: prunedByRegion(e)

Input : e : the entry to be pruned
if $mindist(e, q) > dmax$ **then** `//` $dmax$ `is` $max_{\forall_i}(d_i^k)$
    return true;
for each region P_i that e overlaps with **do**
    if $mindist(e, q) < d_i^k$ **then**
        return false;
return true;

Algorithm 3: updateKthNN(f)

Input : f the facility to be used to update kth-NN

```
Let P_i be the region that contains f;
Insert f in kNN list of P_i;
Set d_i^k to be the distance between q and k-th NN;
Update dmax if required // dmax is max_∀i(d_i^k);
```

Algorithm 4: SIX-Verification

```
Insert root of user R*-tree in a stack h;
while h is not empty do
    pop an entry e from h;
    if not prunedByRegion(e) then // see Algorithm 2
        if e is an intermediate node or leaf then
            insert every child c of e in h ;
        else
            if lessThanKinRange(e) then // see Algorithm 5
                report e as RkNN;
```

Algorithm 5: lessThanKinRange(p)

```
Insert root of facility R*-tree in a min-heap h;
counter ← 0;
range ← dist(p, q);
while h is not empty do
    deheap an entry e;
    if mindist(e, q) > range then
        continue;
    if e is an intermediate node or leaf then
        insert every child c in h with key mindist(q, c);
    else
        counter ← counter + 1;
        if counter >= k then
            return false;
return true;
```

2 TPL

Algorithm 6: TPL-Filtering

Insert root of facility R*-tree in a min-heap h;
Initialize $S_{rfn} \leftarrow \emptyset$ and $S_{fil} \leftarrow \emptyset$;
while h is not empty **do**
    deheap an entry e;
    if kTrim(S_{fil},e)=∞ **then** `// see Algorithm 7`
        $S_{rfn} = S_{rfn} \cup e$;
        continue;
    if e is an intermediate node or leaf **then**
        for each child c of e **do**
            if kTrim(S_{fil},c) $= \infty$ **then** `// see Algorithm 7`
                $S_{rfn} = S_{rfn} \cup c$;
            else
                insert c in h with key $mindist(q, c)$;
    else
        Compute Hilbert value of e;
        Add e to filtering set S_{fil} in ascending order of Hilbert values;

Algorithm 7: kTrim(S_{fil},e)

Input : e : entry/rectangle to be trimmed, S_{fil} : the filtering set in sorted order $\{f_1, \cdots, f_m\}$

$e^{tmp} \leftarrow e$;
while true **do**
  for $i = 1$ to m **do** // m is the size of S_{fil}
    for $j = 0$ to $k - 1$ **do**
      $e^j \leftarrow$ Trim(e^{tmp}, f_{i+j})// Trim e^{tmp} using f_{i+j} and return trimmed entry to e^j (see Clipping algorithm cited in the paper);
    $e^{tmp} = \cup_{j=1}^{k} e^j$;
    if $e^{tmp} = \emptyset$ **then**
      return ∞
  if e^{tmp} is unchanged **then**
    break;
return $mindist(e^{tmp}, q)$

Algorithm 8: TPL-Verification

Insert root of user R*-tree in a stack h;
Initialize $S_{cand} \leftarrow \emptyset$;
while h is not empty **do**
  pop an entry e from h;
  if kTrim(S_{fil},e)=∞ **then** // see Algorithm 7
    continue;
  if e is an intermediate node or leaf **then**
    insert every child c of e in h;
  else
    $S_{cand} = S_{cand} \cup e$;
TPL-Refinement(S_{cand})// see Algorithm 9;

Algorithm 9: TPL-Refinement(S_{cand})

```
Input   : S_cand: candidates set
for each point p in S_cand do
    p.counter ← k;
    for each other point p' in S_cand do
        if dist(p, p') < dist(p, q) then
            p.counter ← p.counter − 1;
    if p.counter = 0 then
        remove p from S_cand;
    else
        p.toVisit ← ∅;
while (true) do
    Refinement_Round_TPL(S_cand ,N_rfn,P_rfn )// see Algorithm 10;
    if S_cand is empty then
        return
    let N be the lowest level node that appears in the largest number of
    p.toVisit for p in S_cand ;
    remove N from all p.toVisit;
    P_rfn ← ∅, N_rfn ← ∅;
    if N is an intermediate node then
        N_rfn ← children of N ;
    else
        P_rfn ← children of N;
```

Algorithm 10: Refinement_Round_TPL(S_{cand} ,N_{rfn},P_{rfn})

```
for each point p in S_cand do
    for each point p' in P_rfn do
        if dist(p, p') < dist(p, q) then
            p.counter ← p.counter − 1;
            if p.counter = 0 then
                remove p from S_cand;
                goto 1;
    for each node N in N_rfn do
        if maxdist(p, N) < dist(p, q) and |N| ≥ p.counter then // |N| is number of points in N
            remove p from S_cand;
            goto 1;
    for each node N in N_rfn do
        if mindist(p, N) < dist(p, q) then
            Add N into p.toVisit;
    if p.toVisit = ∅ then
        remove p from S_cand;
        report p as RkNN;
```

3 TPL++

Algorithm 11: TPL-OPT-Filtering

Insert root of facility R*-tree in a min-heap h;
while h is not empty **do**
  deheap an entry e;
  if e is an intermediate node or leaf **then**
    if isFiltered(S_{fil}, e) = false **then** `// see Algorithm 12`
      $S_{rfn} \leftarrow S_{rfn} \cup e$;
      continue;
    else
      insert every child c in h with key $mindist(q, c)$;
  else
    $S_{fil} \leftarrow S_{fil} \cup e$;

Algorithm 12: isFiltered(S_{fil}, e)

Input : S_{fil}: the filtering set, e: the entry to be filtered
Output : Return true if the entry can be filtered, otherwise return false

counter$\leftarrow 0$;
$e^{tmp} \leftarrow e$;
for each facility $f \in S_{fil}$ **do**
  if e lies completely in $H_{f:q}$ **then**
    $counter \leftarrow counter + 1$;
  else
    $e^{tmp} = \text{Trim}(e^{tmp}, H_{f:q})$;
    if $e^{tmp} = \emptyset$ **then** `// if the whole` e^{tmp} `is pruned`
      $counter \leftarrow counter + 1$;
      $e^{tmp} \leftarrow e$
  if $counter = k$ **then**
    return true
return false

Algorithm 13: TPL-OPT-Verification

Insert root of user R*-tree in a stack h;
Initialize S_{cand} to be empty;
while h is not empty **do**
  pop an entry e from h;
  if isFiltered(S_{fil}, e) **then** `// see Algorithm 12`
    continue;
  if e is an intermediate node or leaf **then**
    insert every child c of e in h;
  else
    $S_{cand} \leftarrow S_{cand} \cup e$;
TPL-Refinement(S_{cand}) `// see Algorithm 9`;

4 FINCH

Algorithm 14: FINCH-Filtering

```
initialize the convex hull CH to the whole data space;
Insert root of facility R*-tree in a min-heap h;
while h is not empty do
    deheap an entry e;
    if overlapsPolygon(e, CH) then // see Algorithm 16
        if e is an intermediate node or leaf then
            for each child c of e do
                if overlapsPolygon(c, CH) then // see Algorithm 16
                    insert c in h with key mindist(q, c);
        else
            UpdateIntersections(e)// see Algorithm 15;
            S_fil ← S_fil ∪ e;
            I_ch ←the left most and right most intersection point on each side of
            data space;
            I_ch ← I_ch ∪ {the intersection points with counter equal to k − 1};
            CH ← convex hull of I_ch;
            compute the minimal bounding rectangle MBR and minimum bounding
            circle MBC of convex hull;
```

Algorithm 15: UpdateIntersections(e)

```
for each existing intersection point i in I do
    if i lies in H_{e:q} then
        i.counter ← i.counter + 1;
        if i.counter >= k then
            remove i from I;
for each facility f in S_{fil} do
    i ←intersection point between H_{f:q} and H_{e:q};
    if i lies inside the MBC of polygon then
        i.counter ← number of facilities in S_{fil} that prune i;
        if i.counter < k then
            insert i in I;
```

Algorithm 16: overlapsPolygon(e, P)

```
Input : e: the entry, P: the polygon
if e does not overlap with the MBC of polygon P then
    return false ;
if e does not overlap with the MBR of polygon P then
    return false ;
if e is a data point then
    if e lies in the polygon P then
        return true;;
else // e is a rectangle
    for each corner point c of e do
        if c lies in the polygon P then
            return true;
    for each corner c of the polygon P do
        if c lies inside the rectangle e then
            return true;
    for each edge X of e do
        for each edge Y of the polygon do
            if X intersects Y then
                return true;
return false;
```

Algorithm 17: FINCH-Verification

```
Insert root of user R*-tree in a stack h;
while h is not empty do
    pop an entry e from h;
    if e is an intermediate node or leaf then
        if overalpsPolygon(e, CH) then // see Algorithm 16
            insert every child c of e in h ;
    else
        if lessThanKinRange(e) then // see Algorithm 5
            report as RkNN;
```

5 InfZone

Algorithm 18: InfZone-Filtering

$V \leftarrow$vertices of the data space;
$r_{max} \leftarrow \infty$;
Insert root of facility R*-tree in a min-heap h;
while h is not empty **do**
    deheap an entry e;
    if prunedByVertices(e, V, r_{max}) = false **then** `// see Algorithm 19`
        if e is an intermediate node or leaf **then**
            insert every child c in h with key $mindist(q, c)$;
        else
            UpdateIntersections(e)`// see Algorithm 15`;
            $V \leftarrow$intersection points with counter equal to $k - 1$;
            $V \leftarrow V \cup \{$intersection points on the boundary of data space$\}$;
            $r_{max} \leftarrow max_{v \in V}(dist(q, v)$;
$V \leftarrow V \cup \{$intersection points with counter equal to $k - 2\}$;
sort vertices in V according to the angle they make with q;
connect adjacent vertices in V to construct the influence zone Z_k;

Algorithm 19: prunedByVertices(e, V, r_{max})

Input : e: entry to be pruned, V: the vertices of the influence zone: r_{max}: $max_{v \in V}(dist(q, v)$
if $mindist(e, q) > 2 \times r_{max}$ **then**
    return true ;
for each vertex $v \in V$ **do**
    if $mindist(e, q) < dist(v, q)$ **then**
        return false;
return true;

Algorithm 20: InfZone-Verification

```
Insert root of user R*-tree in a stack h;
while h is not empty do
    pop an entry e from h;
    if overlapsPolygon(e, Z_k) then // see Algorithm 16
        if e is an intermediate node or leaf then
            insert every child c of e in h ;
        else
            report e as RkNN;
```

6 SLICE

Algorithm 21: SLICE-Filtering

Divide the space around q in t equally sized partitions;
Insert root of facility R*-tree in a min-heap h;
while h is not empty **do**
    deheap an entry e;
    if facilityPruned(e) = false **then** `// see Algorithm 22`
        if e is an intermediate node or leaf **then**
            insert every child c in h with key $mindist(q, c)$;
        else
            pruneSpace(e)`// see Algorithm 23`;
find k-th $UpperArc$ for each partition P;
compute $MinLower$ `//` $MinLower$ `is the minimum lower arc among all partitions`;

Algorithm 22: facilityPruned(e)

Input : e : the entry to be pruned
if $mindist(e, q) > 2 \times MaxUpper$ **then**
    return true;
if $mindist(e, q) < 2 \times MinUpper$ **then**
    return false;
compute the angle range $(\theta_{min}, \theta_{max})$ of e w.r.t q ;
for each partition P that lies within the angle range$(\theta_{min} - 90, \theta_{max} + 90)$ **do**
    if e overlaps with P **then**
        if $mindist(e, q, P) < 2 \times r_P^B$ **then** `//` $mindist(e, q, P)$ `is` $mindist$ `from` q `to the part of` e `that lies in` P
            return false;
    else
        if $mindist(e, A) < r_P^B$ OR $mindist(e, B) < r_P^B$ **then** `//` A `and` B `are the intersection points of arc` r_P^B `with boundaries of` P
            return false;
return true;

Algorithm 23: pruneSpace(f)

```
for each partition P for which minAngle(f, P) < 90° do
    if maxAngle(f, P) ≥ 90° then
        r^U_{f:P} ← ∞ ;
    else
        r^U_{f:P} ← dist(f,q) / (2 cos(maxAngle(f,P)));
    update upper arc list of P;
    set r^B_P to the radius of k-th smallest upper arc of P;
    if f is a significant facility of P then
        insert f in sigList of P in sorted order of lower arc r^L_{f:P};
        // r^L_{f:P} = dist(f,q) / (2 cos(minAngle(f,P)))
Update MaxUpper or MinUpper if required;
```

Algorithm 24: SLICE-Verification

```
Insert root of user R*-tree in a stack h ;
while h is not empty do
    pop an entry e from h;
    if userPruned(e) = false then // see Algorithm 25
        if e is an intermediate node or leaf then
            insert every child c of e in h;
        else
            if isRkNN(e) then // see Algorithm 26
                report e as RkNN;
```

Algorithm 25: userPruned(e)

```
Input : e : the entry to be pruned
if mindist(e, q) > MaxUpper then
    return true;
if mindist(e, q) < MinUpper then
    return false;
compute the angle range(θ_min, θ_max) of e w.r.t q;
for each partition P that lies within angle range(θ_min, θ_max) do
    if mindist(e, q, P) < r^B_P then
        return false;
return true;
```

Algorithm 26: isRkNN(u)

Output : Returns true if u is RkNN. Otherwise, returns false.

if $dist(u, q) < MinLower$ **then**
  return true;
Let P be the partition in which u lies;
if $dist(u, q) \leq r^L_{k:P}$ **then** // $r^L_{k:P}$ is the $k-th$ smallest lower arc in P
  return true;
$counter \leftarrow 0$;
for each $f \in sigList$ of P in ascending order of $r^L_{f:P}$ **do**
  if $dist(u, q) < r^L_{f:P}$ **then**
    return true;
  if $dist(u, f) < dist(u, q)$ **then**
    $counter \leftarrow counter + 1$;
    if $counter \geq k$ **then**
      return false;
return true;