Indici per Query di Similarità

Sistemi informativi per le Decisioni

Slide a cura di Prof. Paolo Ciaccia
Plan of activities

- In the following we will go through 2 distinct topics, all of them being related by the common objective to provide efficient support to the execution of similarity queries

1. We will describe the R-tree, by detailing how to search within a vector space
2. Then, we will consider metric trees, which allow us to deal even with non-vector features and with distance functions other than (weighted) Lp-norms
Can we exploit indices to solve multi-dimensional queries?

- As a first step we consider B+-trees, assuming that we have one multi-attribute index that organizes (sorts) the tuples according to the order $A_1, A_2, ..., A_D$.

- Again, we must understand what this organization implies from a geometrical point of view...
The geometry of B+-trees

- Consider the list of leaf nodes of the B+-tree: N1→N2→N3→…
- The 1st leaf, N1, contains the smallest value(s) of A1, the number of which depends on the maximum leaf capacity C (=2*B+-tree order) and on data distribution
- The 2nd leaf starts with subsequent values, and so on
- The “big picture” is that the attribute space A is partitioned as in the figure

No matter how we sort the attributes, searching for the k-NN of a point q will need to access too many nodes
The basic reason is that “close” points of A are quite far apart in the list of leaves, thus moving along a coordinate (e.g., A1) will “cross” too many nodes

Close points can be here
Another approach based on B+-trees

- Assume that we somehow know, e.g., using DB statistics (see [CG99]), that the k-NN of q are in the (hyper-)rectangle with sides \([l_1,h_1] \times [l_2,h_2] \times \ldots\)
- Then we can issue D independent range queries \(A_i \text{ BETWEEN } l_i \text{ AND } h_i\) on the D indexes on \(A_1, A_2, \ldots, A_D\), and then intersect the results

Besides the need to know the ranges, with this strategy we waste a lot of work
- This is roughly proportional to the union of the results minus their intersection
Multi-dimensional (spatial) indices

The multi-attribute B+-tree maps points of \( A \subseteq \mathbb{R}^D \) into points of \( \mathbb{R} \)

This “linearization” necessarily favors, depending on how attributes are ordered in the B+-tree, one attribute with respect to others

- A B+-tree on \((X,Y)\) favors queries on \(X\), it cannot be used for queries that do not specify a restriction on \(X\)

Therefore, what we need is a way to organize points so as to preserve, as much as possible, their “spatial proximity”

The issue of “spatial indexing” has been under investigation since the 70’s, because of the requirements of applications dealing with “spatial data” (e.g., cartography, geographic information systems, VLSI, CAD)

More recently (starting from the 90’s), there has been a resurrection of interest in the problem due to the new challenges posed by several other application scenarios, such as multimedia

We will now just consider one (indeed very relevant!) spatial index…
The R-tree (Guttman, 1984)

- The R-tree [Gut84] is (somewhat) an extension of the B+-tree to multi-dimensional spaces, in that:
  - The B+-tree organizes objects into
    - a set of (non-overlapping) 1-D intervals,
    - and then applies recursively this basic principle up to the root,
  - the R-tree does the same but now using
    - a set of (possibly overlapping) m-D intervals, i.e., (hyper-)rectangles!,
    - and then applies recursively this basic principle up to the root.

- The R-tree is also available in some commercial DBMS’s, such as Oracle9i
- In the following we just present the aspects relevant to query processing, and postpone the discussion on R-tree management (insertion and split).

Be sure to understand what the index looks like and how it is used to answer queries; for the moment don’t be concerned on how an R-tree with a given structure can be built!
R-tree: the intuition

- Recursive bottom-up aggregation of objects based on MBR’s
- Regions can overlap

This is a 2-D range query using L2, other queries and distance functions can be supported as well.
R-tree basic properties (i)

- The R-tree is a dynamic, height-balanced, and paged tree
- Each node stores a variable number of entries

**Leaf node:**
- An entry $E$ has the form $E=(\text{tuple-key}, \text{TID})$, where tuple-key is the “spatial key” (position) of the tuple whose address is TID (remind: TID is a pointer)

**Internal node:**
- An entry $E$ has the form $E=(\text{MBR}, \text{PID})$, where MBR is the “Minimum Bounding Rectangle” (with sides parallel to the coordinate axes) of all the points reachable from (“under”) the child node whose address is PID (PID = page identifier)

- We can uniform things by saying that each entry has the format $E=(\text{key}, \text{ptr})$
- If $N$ is the node pointed by $E$.ptr, then $E$.key is the “spatial key” of $N$
R-tree basic properties (ii)

- The number of entries varies between $c$ and $C$, with $c \leq 0.5 \times C$ being a design parameter of the R-tree and $C$ being determined by the node size and the size of an entry (in turn this depends on the space dimensionality).

- The root (if not a leaf) makes an exception, since it can have as low as 2 children.

- Note that a (hyper-)rectangle of $\mathbb{R}^D$ with sides parallel to the coordinate axes can be represented using only $2 \times D$ floats that encode the coordinate values of 2 opposite vertices.
Search: range query (i)

- We start with a query type simpler than k-NN queries, namely the

  **Range Query**
  - **Given** a point \( q \), a relation \( R \), a search radius \( r \geq 0 \), and a distance function \( d \),
  - **Determine** all the objects \( t \) in \( R \) such that \( d(t,q) \leq r \)

- The region of \( \mathbb{R}^D \) defined as \( \text{Reg}(q) = \{ p: p \in \mathbb{R}^D, d(p,q) \leq r \} \) is also called the query region (thus, the result is always contained in the query region)
  - For simplicity, both \( d \) and \( r \) are understood in the notation \( \text{Reg}(q) \)

- In the literature there are several variants of range queries, such as:
  - **Point query**: when \( r = 0 \) (i.e., it looks for a perfect (exact) match)
  - **Window query**: when the query region is a (hyper-)rectangle (a window)
Search: range query (ii)

The algorithm for processing a range query is extremely simple:

- We start from the root and, for each entry E in the root node, we check if E.key intersects Reg(q):
  - \(\text{Req}(q) \cap E\text{.key} \neq \emptyset\): we access the child node N referenced by E.ptr
  - \(\text{Req}(q) \cap E\text{.key} = \emptyset\): we can discard node N from the search

- When we arrive at a leaf node we just check for each entry E if \(E\text{.key} \in \text{Reg}(q)\), that is, if \(d(E\text{.key},q) \leq r\).
  - If this is the case we can add E to the result of the index search

\[
\text{RangeQuery}(q,r,N)
\begin{array}{l}
\{ \text{if } N \text{ is a leaf then: for each } E \text{ in } N: } \\
\quad \text{if } d(E\text{.key},q) \leq r \text{ then add } E \text{ to the result} \\
\text{else: for each } E \text{ in } N: } \\
\quad \text{if } \text{Req}(q) \cap E\text{.key} \neq \emptyset \text{ then RangeQuery}(q,r,*(E\text{.ptr})) \}
\end{array}
\]

The recursion starts from the root of the R-tree

- The notation \(N = *(E\text{.ptr})\) means “N is the node pointed by E.ptr”
- Sometimes we also write \(\text{ptr}(N)\) in place of E.ptr
Range queries in action

- The navigation follows a depth-first pattern
- This ensures that, at each time step, the maximum number of nodes in memory is $h=\text{height of the R-tree}$
- Such nodes are managed using a stack
Search: k-NN query (i)

- With the aim to better understand the logic of k-NN search, let us define for a node \( N = *(E.\text{ptr}) \) of the R-tree its region as

\[
\text{Reg}(*(E.\text{ptr})) = \text{Reg}(N) = \{ p : p \in \mathbb{R}^D, p \in E.\text{key}=E.\text{MBR} \}
\]

- Thus, we access node \( N \) if and only if (iff) \( \text{Req}(q) \cap \text{Reg}(N) \neq \emptyset \)

- Let us now define \( d_{\text{MIN}}(q,\text{Reg}(N)) = \inf_p \{d(q,p) \mid p \in \text{Reg}(N)\} \), that is, the minimum possible distance between \( q \) and a point in \( \text{Reg}(N) \).

- The “MinDist” \( d_{\text{MIN}}(q,\text{Reg}(N)) \) is a lower bound on the distances from \( q \) to any indexed point reachable from \( N \).

- We can make the following basic observation:

\[
\text{Req}(q) \cap \text{Reg}(N) \neq \emptyset \iff d_{\text{MIN}}(q,\text{Reg}(N)) \leq r
\]
Search: k-NN query (ii)

- We now present an algorithm, called kNNOptimal [BBK+97], for solving k-NN queries with an R-tree
  - The algorithm also applies to other index structures (e.g., the M-tree) that we will see in this course
- For simplicity, consider the basic case $k=1$
- For a given query point $q$, let $t_{NN}(q)$ be the 1st nearest neighbor (1-NN = NN) of $q$ in $R$, and denote with $r_{NN} = d(q, t_{NN}(q))$ its distance from $q$
  - Clearly, $r_{NN}$ is only known when the algorithm terminates

**Theorem:**
- Any algorithm for 1-NN queries must visit at least all the nodes $N$ whose MinDist is less than $r_{NN}$

**Proof:** Assume that an algorithm ALG stops by reporting as NN of $q$ a point $t$ and that ALG does not read a node $N$ such that (s.t.) $d_{MIN}(q, \text{Reg}(N)) < d(q,t)$; then $\text{Reg}(N)$ might contain a point $t'$ s.t. $d(q,t') < d(q,t)$, thus contradicting the hypothesis that $t$ is the NN of $q$
The logic of the kNNOptimal Algorithm

- The kNNOptimal algorithm uses a priority queue $PQ$, whose elements are pairs $[\text{ptr}(N), d_{\text{MIN}}(q,\text{Reg}(N))]$.
- $PQ$ is ordered by increasing values of $d_{\text{MIN}}(q,\text{Reg}(N))$.
  - $\text{DEQUEUE}(PQ)$ extracts from $PQ$ the pair with minimal MinDist.
  - $\text{ENQUEUE}(PQ, [\text{ptr}(N), d_{\text{MIN}}(q,\text{Reg}(N))])$ performs an ordered insertion of the pair in the queue.
- Pruning of the nodes is based on the following observation:
  - If, at a certain point of the execution of the algorithm, we have found a point $t$ s.t. $d(q,t) = r$,
  - Then, all the nodes $N$ with $d_{\text{MIN}}(q,\text{Reg}(N)) \geq r$ can be excluded from the search, since they cannot lead to an improvement of the result.

- In the description of the algorithm, the pruning of pairs of $PQ$ based on the above criterion is concisely denoted as $\text{UPDATE}(PQ)$.
- With a slight abuse of terminology, we also say that “the node $N$ is in $PQ$” meaning that the corresponding pair $[\text{ptr}(N), d_{\text{MIN}}(q,\text{Reg}(N))]$ is in $PQ$.
- Intuitively, kNNOptimal performs a “range search with a variable (shrinking) search radius” until no improvement is possible anymore.
The kNNOptimal Algorithm (case k=1)

Input: query point q, index tree with root node RN
Output: $t_{NN}(q)$, the nearest neighbor of q, and $r_{NN} = d(q, t_{NN}(q))$

1. Initialize PQ with [ptr(RN),0]; // starts from the root node
2. $r_{NN} := \infty$; // this is the initial “search radius”
3. while PQ ≠ ∅: // until the queue is not empty…
4. $[ptr(N), d_{MIN}(q,\text{Reg}(N))] := \text{DEQUEUE}(PQ)$; // … get the closest pair…
5. Read(N); // … and reads the node
6. if N is a leaf then: for each point t in N:
7. if $d(q,t) < r_{NN}$ then: {$t_{NN}(q) := t; r_{NN} := d(q,t); \text{UPDATE}(PQ)$} // reduces the search radius and prunes nodes
8. else: for each child node Nc of N:
9. if $d_{MIN}(q,\text{Reg}(Nc)) < r_{NN}$ then:
10. $\text{ENQUEUE}(PQ, [ptr(Nc), d_{MIN}(q,\text{Reg}(Nc))]);$
11. return $t_{NN}(q)$ and $r_{NN}$;
12. end.
kNNOptimal in action

- Nodes are numbered following the order in which they are accessed.
- Objects are numbered as they are found to improve (reduce) the search radius.
- The accessed leaf nodes are shown in red.

Indici per query di similarità
Correctness and Optimality of kNNOptimal

- The kNNOptimal algorithm is clearly correct
- To show that it is also optimal, that is, it reads the minimum number of nodes, it is sufficient to prove that it never reads a node \( N \) s.t. \( d_{\text{MIN}}(q, \text{Reg}(N)) > r_{NN} \)

**Proof:**

- Indeed, \( N \) is read only if, at a certain execution step, it becomes the 1st element in the priority queue \( PQ \)
- Let \( N_1 \) be the node containing \( t_{NN}(q) \), \( N_2 \) its parent node, \( N_3 \) the parent node of \( N_2 \), and so on, up to \( N_h = R_N \) (\( h = \) height of the tree)
- Now observe that, by definition of MinDist, it is:
  \[ r_{NN} \geq d_{\text{MIN}}(q, \text{Reg}(N_1)) \geq d_{\text{MIN}}(q, \text{Reg}(N_2)) \geq \ldots \geq d_{\text{MIN}}(q, \text{Reg}(N_h)) \]
- At each time step before we find \( t_{NN}(q) \), one (and only one) of the nodes \( N_1, N_2, \ldots, N_h \) is in the priority queue
- It follows that \( N \) can never become the 1st element of \( PQ \)
The general case \( (k \geq 1) \)

- The algorithm is easily extended to the case \( k \geq 1 \) by using:
  - a data structure, which we call ResultList, where we maintain the \( k \) closest objects found so far, together with their distances from \( q \)
  - as “current search radius” the distance, \( r_{k-NN} \), of the current \( k \)-th NN of \( q \), that is, the \( k \)-th element of ResultList

<table>
<thead>
<tr>
<th>ObjectID</th>
<th>distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>t15</td>
<td>4</td>
</tr>
<tr>
<td>t24</td>
<td>8</td>
</tr>
<tr>
<td>t18</td>
<td>9</td>
</tr>
<tr>
<td>t4</td>
<td>12</td>
</tr>
<tr>
<td>t2</td>
<td>15</td>
</tr>
</tbody>
</table>

- The rest of the algorithm remains unchanged
The kNNOptimal Algorithm (case $k \geq 1$)

**Input:** query point $q$, integer $k \geq 1$, index tree with root node RN

**Output:** the $k$ nearest neighbors of $q$, together with their distances

1. Initialize PQ with $[\text{ptr}(RN),0]$;
2. for $i=1$ to $k$: ResultList$(i) := [\text{null}, \infty]$; $r_{k-\text{NN}} := $ ResultList$(k).\text{dist}$;
3. while $\text{PQ} \neq \emptyset$:
4.   $[\text{ptr}(N), d_{\text{MIN}}(q,\text{Reg}(N))] := \text{DEQUEUE}(\text{PQ})$;
5.   Read$(N)$;
6.   if $N$ is a leaf then: for each point $t$ in $N$:
7.     if $d(q,t) < r_{k-\text{NN}}$ then: { remove the element in ResultList$(k)$;
8.       insert $[t,d(q,t)]$ in ResultList;
9.       $r_{k-\text{NN}} := $ ResultList$(k).\text{dist}$; \text{UPDATE}(\text{PQ})}
10. else: for each child node $N_c$ of $N$:
11.    if $d_{\text{MIN}}(q,\text{Reg}(N_c)) < r_{k-\text{NN}}$ then:
12.       ENQUEUE$(\text{PQ},[\text{ptr}(N_c), d_{\text{MIN}}(q,\text{Reg}(N_c))])$;
13. return ResultList;
14. end.
Back to the R-tree

- It’s now time to discuss how an R-tree can be effectively built.
- It has to be considered that many “R-tree variants” exist, and it’s not our intention to go through their details.
- It just suffices to say that one of such variants leads to what is known as the R*-tree [BKS+90], which is the commonest version in use.
- With respect to the original proposal [Gut84], the R*-tree adds smarter insertion and split heuristics, plus a so-called “forced reinsert” technique that we do not consider here.
R-tree: how it looks like

Remind:
- Recursive bottom-up aggregation of objects based on MBR’s
- Regions can overlap
- Each node can contain up to \( C \) entries, but not less than \( c \leq 0.5 \times C \)
  - The root makes an exception
R-tree: insertion of a new object

- We start from the root and move down the tree one step at a time, trying to find a “nice place” where to accommodate the new object \( p \).
- For simplicity, we assume that indexed objects are points, similar arguments apply if we index (hyper-)rectangles (MBR's).
- At each step we have a same question to answer: Which child node is the most suitable to accommodate \( p \)?

Which child node is the most suitable to accommodate \( p \)?

And here?
R-tree: the ChooseSubtree method

- The recursive algorithm that descends the tree to insert a new object $p$, together with its TID, is called ChooseSubtree

```python
ChooseSubtree(Ep=(p,TID),ptr(N))
1. Read(N);
2. If N is a leaf then: return N          // we are done
3. else: { choose among the entries Ec in N
           the one, Ec*, for which Penalty(Ep,Ec*) is minimum;
           return ChooseSubtree(Ep,Ec*.ptr) }       // recursive call
4. end.
```

- We invoke the method on the index root
- The specific criterion used to decide “how bad” an entry is, should we choose it to insert $p$, is encapsulated in the Penalty method
  - Variants of the R-tree differ in how they implement Penalty
- This insertion algorithm is the one used by most multi-dimensional and metric trees
R-tree: the Penalty method

- If point \( p \) is inside the region of an entry \( E_c \), then the penalty is 0.
- Otherwise, Penalty can be computed as the increment of volume (area) of the MBR.
  - However, if \( E_c \) points to a leaf node, then \([BKS+90]\) shows that it’s better to consider the increment of overlap with the other entries.

- Both criteria aim to obtain trees with better performance:
  - **Large area**: increases the number of nodes to be visited by a query.
  - **Large overlap**: also degrades performance.

![Diagram showing A and B with point p, and B is better than A.](image)
R-tree: splitting of a leaf node

- When \( p \) has to be inserted into a leaf node that already contains \( C \) entries, an overflow occurs, and \( N \) has to be split.
- For leaf nodes whose entries are points the solution aims to split the set of \( C+1 \) points into 2 subsets, each with at least \( c \) and at most \( C \) points.
- Among the several possibilities, one could consider the choice that leads to have a minimum overall area.
  - However, this is an NP-Hard problem, thus heuristics have to be applied.

\[
\begin{array}{c}
\text{\( C = 16 \)} \\
\text{\( c = 6 \)}
\end{array}
\]
R-tree: splitting of a non-leaf node

- As in B+-trees, splits propagate upward and can recursively trigger splits at higher levels of the tree
- The problem to be faced now is how to split a set of C+1 (hyper-)rectangles
  - Note that this applies also to leaf nodes if they store MBR’s
- The original proposal just aims to minimize the sum of resulting areas
- The R*-tree implements a more sophisticated criterion, which takes into account the areas, overlap, and perimeters of the resulting regions
Beyond vector spaces

- It’s a matter of fact that vector spaces, equipped with some (weighted) Lp-norm, are not general enough to deal with the whole variety of feature types and distance functions needed in MMDB’s.

Example:

given 2 sets of points s1 and s2, their Hausdorff distance is defined as follows:

1. ∀ (red) point of s1 find the closest (blue) point in s2
   Let \( h(s1,s2) \) be the maximum of such distances

2. ∀ (blue) point in s2 find the closest (red) point in s1
   Let \( h(s2,s1) \) be the maximum of such distances

3. Let \( d_{\text{Haus}}(s1,s2) = \max\{ h(s1,s2), h(s2,s1) \} \)

Used for matching shapes
Another example: edit distance

A common distance measure for *strings* is the so-called edit distance, defined as the minimum number of characters that have be inserted, deleted, or substituted so as to transform a string \( s_1 \) into another string \( s_2 \)

\[
d_{\text{edit}}('ball','bull') = 1 \\
d_{\text{edit}}('balls','bell') = 2 \\
d_{\text{edit}}('rather','alter') = 3
\]

The edit distance is also commonly used in *genomic DB’s* to compare DNA sequences. Each DNA sequence is a string over the 4-letters alphabet of bases:

- a: adenine
- c: cytosine
- g: guanine
- t: thymine

\[
d_{\text{edit}}('gatctggtgtg','agcaaatcag') = 7
\]

The edit distance can be computed using a dynamic programming procedure.
Metric spaces

- A metric space \( M = (U,d) \) is a pair, where
  - \( U \) is a domain ("universe") of values, and
  - \( d \) is a distance function that, \( \forall x,y,z \in U \), satisfies the metric axioms:

\[
\begin{align*}
  d(x,y) &\geq 0, \ d(x,y) = 0 \iff x = y \quad \text{(positivity)} \\
  d(x,y) &= d(y,x) \quad \text{(symmetry)} \\
  d(x,y) &\leq d(x,z) + d(z,y) \quad \text{(triangle inequality)}
\end{align*}
\]

- All the distance functions seen in the previous examples are metrics, and so are the (weighted) \( L_p \)-norms

 Metric indexes only use the metric axioms to organize objects, and exploit the triangle inequality to prune the search space
Principles of metric indexing (i)

- Given a “metric dataset” $P \subseteq U$, one of the two following principles can be applied to partition it into two subsets.

**Ball decomposition**: take a point $v$ ("vantage point"), compute the distances of all other points $p$ w.r.t. $v$, $d(p,v)$, and define

$$P_1 = \{ p : d(p,v) \leq r_v \} \quad P_2 = \{ p : d(p,v) > r_v \}$$

If $r_v$ is chosen so that $|P_1| \approx |P_2| \approx |P|/2$ we obtain a balanced partition.

Consider a range query $\{ p : d(p,q) \leq r \}$
If $d(q,v) > r_v + r$ we can conclude that no point in $P_1$ belongs to the result.

**Proof**:
we show that $d(p,q) > r$ holds $\forall p \in P_1$.

$$d(p,q) \geq d(q,v) - d(p,v) \quad \text{(triangle ineq.)}$$

$$> r_v + r - d(p,v) \quad \text{(by hyp.)}$$

$$\geq r_v + r - r_v \quad \text{(by def. of P1)}$$

$$\geq r$$

Similar arguments can be applied to $P_2$.
Principles of metric indexing (ii)

**Generalized Hyperplane:** take two points \( v_1 \) and \( v_2 \), compute the distances of all other points \( p \) w.r.t. \( v_1 \) and \( v_2 \), and define

\[
P_1 = \{ p : d(p,v_1) \leq d(p,v_2) \} \quad P_2 = \{ p : d(p,v_2) < d(p,v_1) \}
\]

Consider a range query \( \{ p : d(p,q) \leq r \} \)

If \( d(q,v_1) - d(q,v_2) > 2* r \) we can conclude that no point in \( P_1 \) belongs to the result

**Proof:**
we show that \( d(p,q) > r \) holds \( \forall p \in P_1 \).

\[
d(q,v_1) - d(p,q) \leq d(p,v_1) \quad \text{(triangle ineq.)}
d(p,v_1) \leq d(p,v_2) \quad \text{(def. of P1)}
d(p,v_2) \leq d(p,q) + d(q,v_2) \quad \text{(triangle ineq.)}
\]

Then:

\[
d(q,v_1) - d(p,q) \leq d(p,q) + d(q,v_2)
d(p,q) \geq (d(q,v_1) - d(q,v_2))/2
\]

\[
> r \quad \text{(by hyp.)}
\]
The M-tree (Ciaccia, Patella, Zezula, 1997)

- The M-tree has been the first dynamic, paged, and balanced metric index.
- Intuitively, it generalizes “R-tree principles” to arbitrary metric spaces.
  - The M-tree treats the distance function as a “black box”.
- Since 1997 [CPZ97] it has been used by several research groups for:
  - Image retrieval, text indexing, shape matching, clustering algorithms (including the WWW log example), fingerprint matching, DNA DB’s, etc.
  - [CNB+01] and [HS03] are both excellent surveys on searching in metric spaces.
- C++ source code freely available at http://www-db.deis.unibo.it/Mtree/

Remind: at a first sight, the M-tree “looks like” an R-tree. However, remember that the M-tree only “knows” about distance values, thus it ignores coordinate values and does not rely on any “geometric” (coordinate-based) reasoning.
M-tree: how it looks like

- Recursive bottom-up aggregation of objects based on regions
- Regions can overlap
- Each node can contain up to \( C \) entries, but not less than \( c \leq 0.5 \times C \)
  - The root makes an exception

Depending on the metric, the “shape” of index regions changes

- \( \text{L}_1 \)
- \( \text{L}_\infty \)
- Weighted Euclidean
- Quadratic distance
The M-tree regions

- Each node $N$ of the tree has an associated region, $\text{Reg}(N)$, defined as
  \[ \text{Reg}(N) = \{ p : p \in U, \, d(p, v_N) \leq r_N \} \]
  where:
  - $v_N$ (the “center”) is also called a routing object, and
  - $r_N$ is called the (covering) radius of the region

- The set of indexed points $p$ that are reachable from node $N$ are guaranteed to have $d(p, v_N) \leq r_N$

- This immediately makes it possible to apply the pruning principle:
  If $d(q, v_N) > r_N + r$ then prune node $N$:  

\[ r_N \quad v_N \quad p \]
Entries of leaf and internal nodes

- Each node $N$ stores a variable number of *entries*

**Leaf node:**
- An entry $E$ has the form $E = (\text{ObjFeatures}, \text{distP}, TID)$, where
  - *ObjFeatures* are the feature values of the indexed object
  - *distP* is the distance between the object and its parent routing object (i.e., the routing object of node $N$)

**Internal node:**
- An entry $E$ has the form $E = (\text{RoutingObjFeatures}, \text{CoveringRadius}, \text{distP}, \text{PID})$, where
  - *RoutingObjFeatures* are the feature values of the routing object
  - *CoveringRadius* is the radius of the region
  - *distP* is the distance between the routing object and its parent routing object (this is undefined for entries in the root node)
Entries: an example

\[(2,3), 2, p1\]

\[N^7\]

\[(2,5), 2.5, \sqrt{5}, \_\]

\[N^3\]

\[(4,6), 5, \_, \_\]
Fast pruning based on distP

- Pre-computed distances distP are exploited during query execution to save distance computations
- Let $v_P$ be the parent (routing) object of $v_N$
- When we come to consider the entry of $v_N$, we
  - have already computed the distance $d(q,v_P)$ between the query and its parent
  - know the distance $d(v_P,v_N)$

From the triangle inequality it is:

$$d(q,v_N) \geq |d(q,v_P) - d(v_P,v_N)|$$

Thus we can prune node $N$ without computing $d(q,v_N)$ if

$$|d(q,v_P) - d(v_P,v_N)| > r_N + r$$
Example (edit distance)

query = “spire”, r = 1

d(“spire”, “shakespeare”) = 3 ≤ 5 + 1

| d(“spire”, “parse”) – d(“parse”, “spare”) | = | 3 – 0 | = 3 ≤ 3 + 1

| d(“spire”, “spare”) – d(“pier”, “spare”) | = | 1 – 4 | = 3 > 1 + 1

| d(“spire”, “spare”) – d(“parse”, “spare”) | = | 1 – 2 | = 1 ≤ 3 + 1

| d(“spire”, “parse”) | = 3 ≤ 3 + 1

Indici per query di similarità