**Distance Matching algorithm:**

To answer a pattern Query we can have two approaches:

The approaches consider two aspects:

1. Centroid or Start based distances

2. Breadth-first or depth-first navigation

**Centroid-based distances:** In this approach, each star is a potential centroid of a shape. Initial matching can be applied between the query centroid and the potential centroid of a shape.

* Centroid (centroid\_id, <<d1,starid\_1,ra\_1,dec\_1>,…,<dn,starid\_n,ra\_n,dec\_n>>)
* Query-centroid (centroid\_id, <<d1,q\_1 >, …,<dk,q\_k >>)
* Query\_star (query\_id, <<d1,q\_1 >,<d2,q\_2 >,…,<dk,q\_k >>)

**Star-based distances:** In this approach, each star may match with any of the query objects, composing the shape.

* Star (starid, <<d1,starid\_1,ra\_1,dec\_1>,..,<dn,starid\_n,ra\_n,dec\_n>>)
* Query (q\_id, <<d1,q\_1 >,<d2,q\_2 >,…,<dk,q\_k >>)
* *Star* Approach algorithm:

*-----------------------------------------------------------------------------------------------------------------------------------*

*For each s in Star*

*For each q in Query*

*M= approx\_match (s,q)*

*/\* finds sets of stars, s.t. for all di \in q exists s.dj in s and approx(s.dj, q.di) \*/*

*if (M is not empty)*

 *for each m in M*

 *for each m.star*

 *Match\_candidate (s, m.id,m.star, mi)*

 */\* stores matches by distance \*/*

 *endFor*

 *endFor*

 *endFor*

*endFor*

*-----------------------------------------------------------------------------------------------------------------------------------*

**Depth-first (Adapted to Star Approach) algorithm:**

*-----------------------------------------------------------------------------------------------------------------------------------*

*match\_shape (Star, Q) {*

*for each s in Star*

*If (s not in Match\_candidate)*

 *evaluate\_match (s,Q)*

*else*

 *U=find\_q\_inMatch (Match, s)*

 *Q=Q-U /\* get query elements already matched with s \*/*

 *evaluate\_match (s,Q)*

*endFor*

*}*

*evaluate\_match (s, Q) {*

*for each q in Q*

*M= approx\_match (s,q) /\* finds sets of stars, s.t. for all di \in q exists s.dj in s and approx(s.dj, q.di) \*/*

*if (M is not empty)*

*for each m in M*

*for each m.star*

 *test=match\_positioned (m.star,m,mi)*

 *if (test)*

 *Match\_candidate (s,m.id,m.star, mi)*

*/\* stores matches by distance \*/*

 *endFor*

 *endFor*

 *endif*

*endFor*

*}*

*boolean match\_positioned (s,m,qi) {*

 *return match (s,m,qi)*

*}*

*-----------------------------------------------------------------------------------------------------------------------------------*

1. **Pre-computing the distances of every sky object from other objects in the dataset in a specific boundary:**

We pre-compute the distances of every sky object from all other objects in a specific boundary of it and store an ordered list of distances and points in a boundary **d-radius** far from the correspondent sky object (we can also use quadtrees or octrees for this purpose). Different stars can have different number of points in their d-radius neighborhood and **we keep all of them.** Here is the result of this step:

d

star-id

**Catalogue-partners**(centroid-star-id, list of <star-id, distance, ra, dec>)

|  |
| --- |
| **Catalogue-partners** |
| **Centroid star-id** | **list of partners in the boundary****(in ascending distance order)** |
| ... | ... |
| s13 | **<s15,d13\_15, s15-ra, s15-dec>, <s18, d13\_15,s18-ra, s18-dec>,** **<s19,d13\_19, s19-ra, s19-dec>, <s21, d13\_21, s21-ra, s21-dec>** |
| s14 | **<s10, d14\_10, s10-ra, s10-dec>, …, <sm, d14\_m, sm-ra, sm-dec>** |
| ... | **...** |

1. **Pre-compute the Query:**
* Find the centroid point in the query.
* Identify some small number of points k' among q1 to qk that are far apart from one another. We want those because the error matters less when points are far apart. Then, we perform the join-style algorithm on the sets corresponding to those k' points and then for each candidate group of stars that survive that filter, test all possible pairs.

For example in the following query, we ignore the points (q4, q5, q6, q7) in the first round and process only (q0, q1, q2, q3).



* Find the most k representative points in the query (it should be validated within real astronomy queries).
	+ Representative elements give more value to our pattern query in a special way. In every area, we should define them differently. (For example in time series maybe those which are more oscillated are more important and representative - In astronomy those who are on the contours of the pattern query or bigger or brighter or more critical in the shape of pattern query, etc.)
	+ Computing the pattern query based on representative points free us from many computations. For example in a query with 50% of points very similar to each other, we can avoid from 50% of the computations in whole.
	+ In every application we need to define how a query element can be more representative and give more value to our pattern query.
	+ Representative points (or seed points), can be useful in the case we don't find complete solutions in the dataset (complete solution is a solution with k points, k= the query size); so we can decide if we return null to the user or we look for a derived pattern query with all seed points.
	+ To rank between a complete solution with all k points but far apart points and a solution with k-2 points which contains all the seed points and has very similar pairwise distances corresponding to the query points, the importance of seed points can be decisive.
1. **Calculate the pairwise query points distances:**

We compute **k(k-1)/2** pairwise distances in the query( k is the number of points in the query) and then choosing the centroid we sort the distances in ascending order and keep the pairwise distances in the following relation:

**Query-pairwise-distances** (q-id-row, q-id-col, distance)

// Centroid is always at id = 0

For example in a query with k = 4 points which q0 is the centroid, we have:



|  |
| --- |
| **Query-pairwise-distances** |
| **q-id-row** | **q-id-col** | **distance** |
| 0 | 1 | d0 |
| 0 | 2 | d4 |
| 0 | 3 | d3 |
| 1 | 3 | d5 |
| 1 | 2 | d1 |
| 2 | 3 | d2 |

1. **Find the candidate matches:**

We look for candidate matches with respect to the query centroid and the result would be stored in the following relation. Every centroid could have a set of candidates for p-th element in the query.

**Cp (Bucket p)**

(centroid\_star\_id, partner\_star\_id, Ra\_partner, Dec\_partner)

|  |
| --- |
| **C1 (Bucket 1: all matches for q0q1)** |
| **centroid\_star\_id** | **partner\_star-id** | **Ra\_partner** | **Dec\_partner** |
| s12 | s15 | ... | ... |
| s12 | s22 |
| s12 | s24 |
| s12 | s15 |
| s12 | s18 |
| s12 | s42 |
| s35 | s18 |
| s46 | s18 |
| ... |  |  |  |

and we would have different tables for C2 (matches for q0q2), C3, ...

1. **Join operation (Build the candidate solutions)**
2. **Dennis’s approach**:

assign an order to the stars in each Cp.

Create a set of matrices Mij with a 1 in position a,b if star a in Ci is close to distance dist(qi,qj) with respect to star b of Cj.

Now if we create a matrix Mjr for Cj, Cr with respect to dist(qj, qr) and multiply these two matrices together, then a 1 in location a,c means that there is some b such that Mij(a,b)=1 and Mjr(b,c) = 1.

The relation formulation is similar we would have an entry in Mij(icol,jcol) for a,b and an entry in Mjr(jcol,rcol) for b,c. If we do a join of the two, then we get select \* from Mij, Mjr where Mij.jcol = Mjr.jcol

ex: Let's say we have matrices i, j, r, s. Then the query would look like

**select \* from Mij, Mjr, Mrs, Msi**

**where Mij.jcol = Mjr.jcol**

**and Mjr.rcol = Mrs.rcol**

**and Mrs.scol = Msi.scol**

**and Msi.icol = Mij.icol**

The resulting 4-tuples then have to be checked against **Mir, Mjs**

In the following figure, we show a query with 5 points and the set of candidates in (Cp)s and Mij matrices:

q0

qi

qj

qr

qs

Cj

Cr

Cs

Ci

d

Mrs

In other words, the join process to find the candidate solutions is done in three steps as following:

Ci

Cj

Cr

Cs

Mij

Mjr

Mrs

Msi

Ci

Mir

Mjs

Mij

solutions

1. **Fabio’s approach join operations:**

Ci

Cj

Cr

Cs

select \* from Ci

**join Cj where** Ci.centroid-id=Cj.centroid-id **and** match( ([Ci.ra\_partner]-[Cj.ra\_partner]),

 ([Ci.dec\_partner]-[Cj.dec\_partner]), q-id-row[i], q-id-col[j])

**join Cr where** Ci.centroid-id=Cr.centroid-id **and** match( ([Ci.ra\_partner]-[Cr.ra\_partner]),

 ([Ci.dec\_partner]-[Cr.dec\_partner]), q-id-row[i], q-id-col[r]) **and**

 match( ([Cj.ra\_partner]-[Cr.ra\_partner]),

([Cj.dec\_partner]-[Cr.dec\_partner]), q-id-row[j], q-id-col[r])

**join Cs where** Ci.centroid-id=Cs.centroid-id **and** match( ([Ci.ra\_partner]-[Cs.ra\_partner]),

 ([Ci.dec\_partner]-[Cs.dec\_partner]), q-id-row[i], q-id-col[s]) **and**

 match( ([Cj.ra\_partner]-[Cs.ra\_partner]),

 ([Cj.dec\_partner]-[Cs.dec\_partner]), q-id-row[j], q-id-col[s]) **and**

 match( ([Cr.ra\_partner]-[Cs.ra\_partner]),

 ([Cr.dec\_partner]-[Cs.dec\_partner]), q-id-row[r], q-id-col[s])

1. **Ranking the solutions:**

We rank ascendingly the solutions by sum of the errors in their distances:

<s12, s15, s19, s33>, Total-cost = 21.5

<s12, s18, s19, s33>, Total- cost = 33

<s2, s18, s21, s35>, Total- cost = 33.5