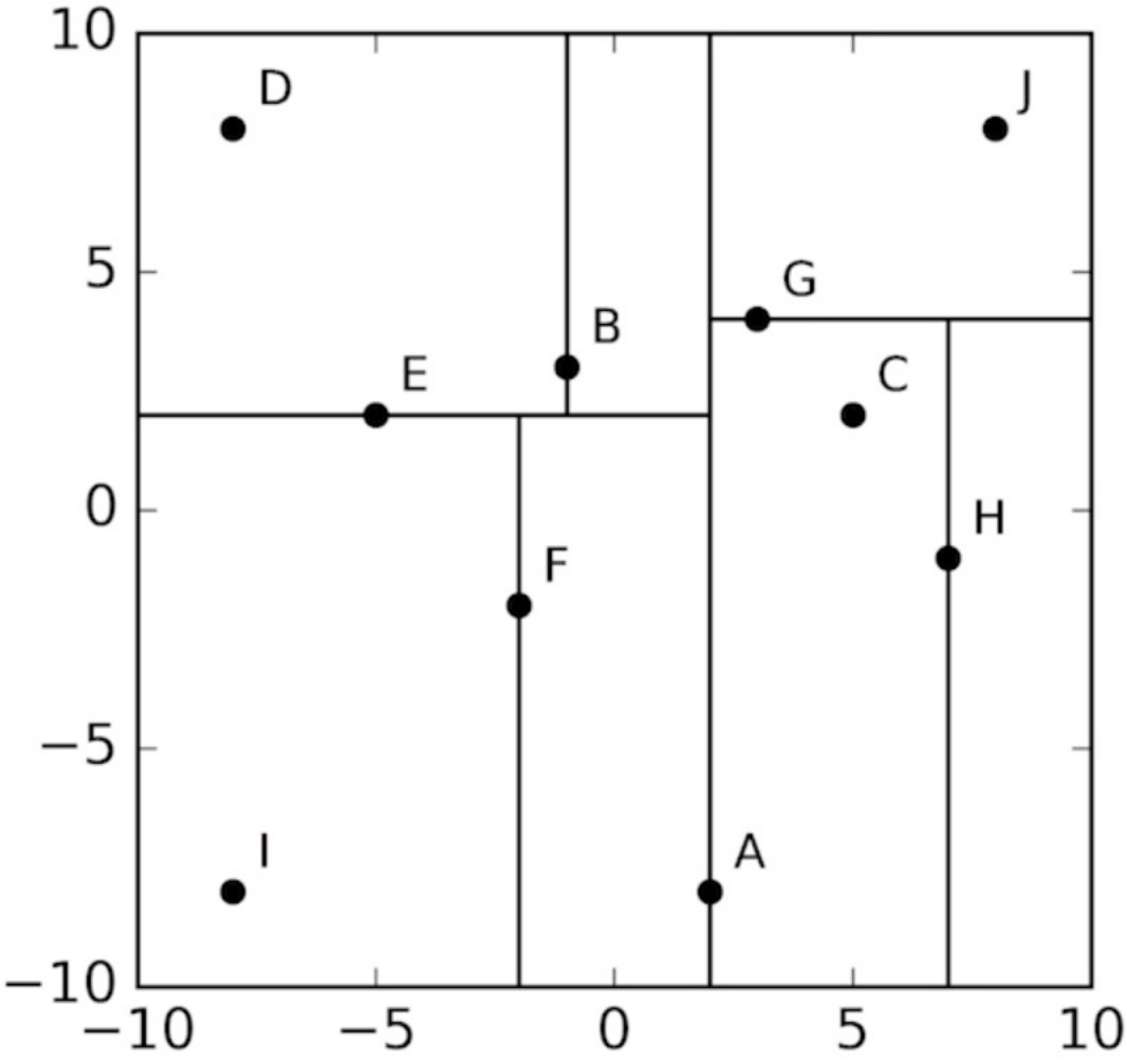
Astropy's SkyCoord Function

K-Dimension Tree Approach for crossmatching

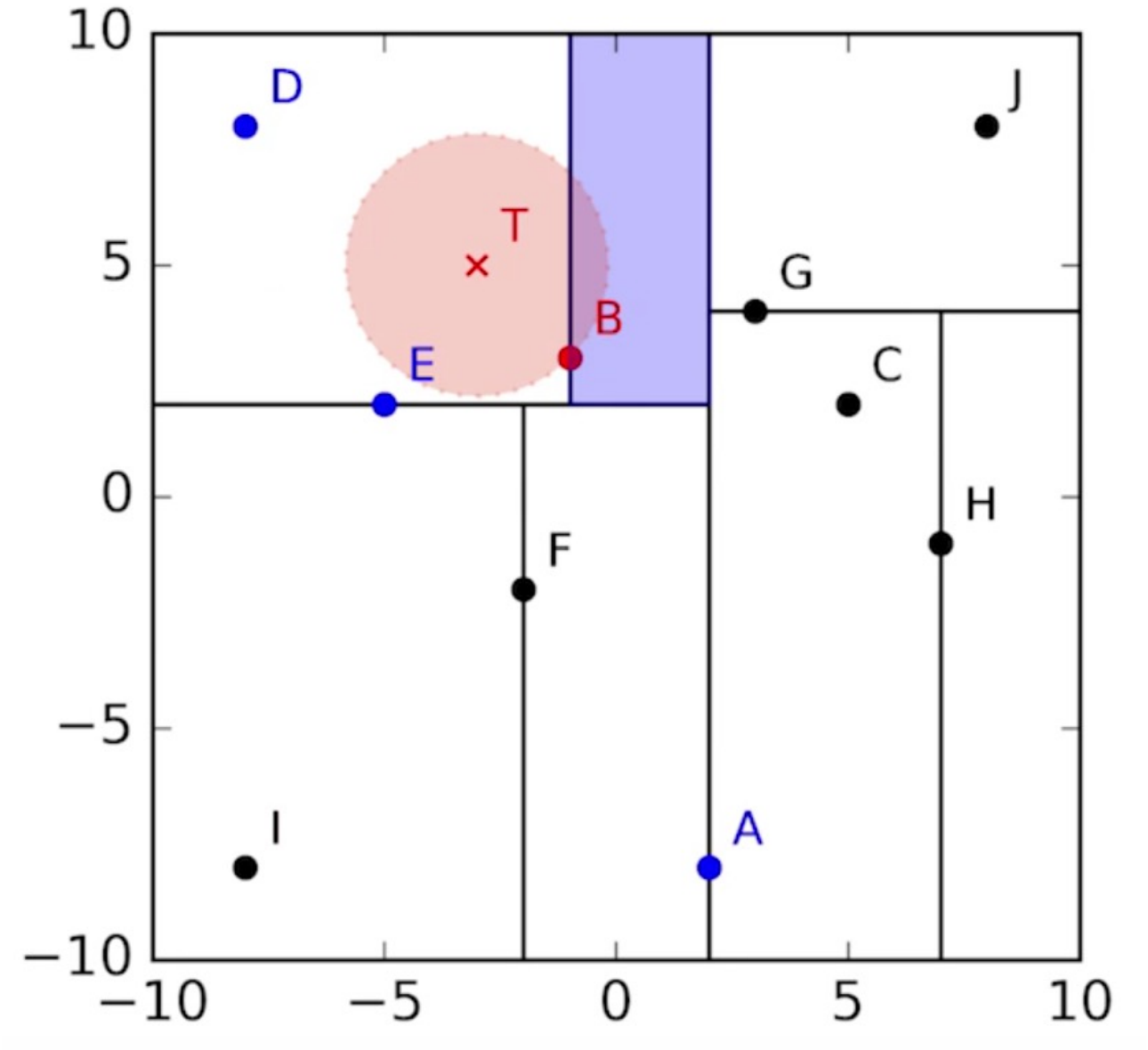
crossmatching is a very common task in astrophysics, so it's natural that it's had optimised implementations written of it already. A popular implementation in Python is found in the **Astropy** module and it uses objects called **k-d trees** to perform crossmatching incredibly quickly.

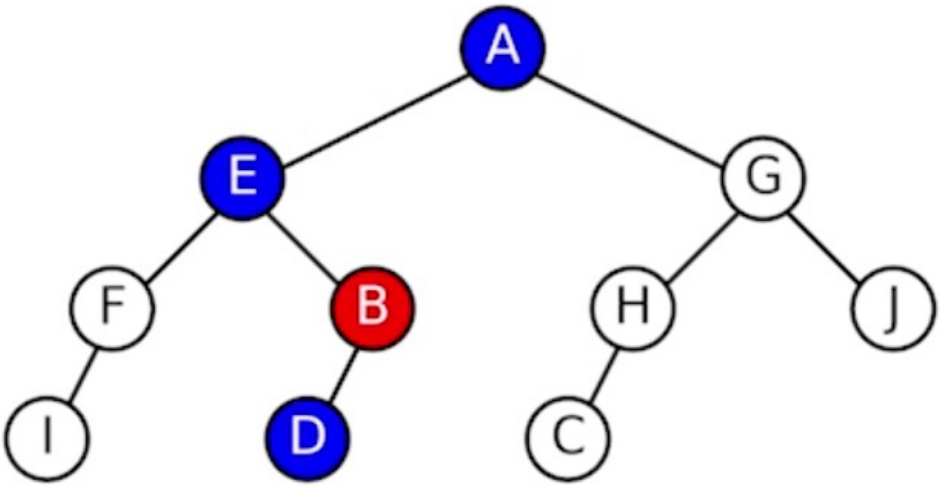


Astropy constructs a k-d tree out of the second catalogue, letting it search through for a match for each object in the first catalogue efficiently. Constructing a k-d tree is similar to the binary search you saw earlier. The k-dimensional space is divided into two parts recursively until each division only contains only a single object. Creating a k-d tree from an astronomy catalogue works like this:

1. Find the object with the median right ascension, split the catalogue into objects left and right partitions of this
2. Find the objects with the median declination in each partition, split the partitions into smaller partitions of objects down and up of these
3. Find the objects with median right ascension in each of the partitions, split the partitions into smaller partitions of objects left and right of these
4. Repeat 2-3 until each partition only has one object in it

This creates a binary tree where each object used to split a partition (a node) links to the two objects that then split the partitions it has created (its children).





Once you've made a k-d tree out of a catalogue, finding a match to an object then works like this:

1. Calculate the distance from the object to highest level node (the root node), then go to the child node closest (in right ascension) to the object
2. Calculate the distance from the object to this child, then go to the child node closest (in declination) to the object
3. Calculate the distance from the object to this child, then go to the child node closest (in right ascension) to the object
4. Repeat 2-3 until you reach a child node with no further children (a leaf node)
5. Find the shortest distance of all distances calculated, this corresponds to the closest object

Since each node branches into two children, a catalogue of N objects will have, on average, log2(N) nodes from the root to any leaf. So while it seems like a lot of effort to create a k-d tree, doing so lets you, for example, search the entire SuperCOSMOS catalogue of 250 million objects using only 28 distance calculations.

Here's an example of using Astropy to crossmatch two catalogues with 2 objects each:

```
from astropy.coordinates import SkyCoord
from astropy import units as u
coords1 = [[270, -30], [185, 15]]
coords2 = [[185, 20], [280, -30]]
sky_cat1 = SkyCoord(coords1*u.degree,
frame='icrs')
sky_cat2 = SkyCoord(coords2*u.degree,
frame='icrs')
closest_ids, closest_dists, closest_dists3d =
sky_cat1.match_to_catalog_sky(sky_cat2)
print(closest_ids)
print(closest_dists)
```

The `SkyCoord` objects are general purpose sky catalogue storage and manipulation objects in Astropy. They take anything that looks like an array of coordinates as long as you specify the units (here we specify degrees with **`u.degree`**) and a reference frame (**ICRS** is essentially the same as equatorial coordinates. The outputs, `closest_id` and `closest_dists` give the matching object's row index in `sky_cat2` and the distance to it. `closest_dists` is the angular distance while `closest_dists3d` is the 3D distance which we're not concerned with here.

Note

Astropy returns distances as `Quantity` objects. You can convert these to NumPy arrays by accessing their `value` attribute like this:
`closest_dists_array = closest_dists.value`