**Module 4. Pairing the DP pairs from HPO and finding out the shared DP in each pair**

/\* To assemble the DDSN-C, the most obvious way is to start from the D, by self-joining the D pairs (in bipartite) to find (i) all the pairs of D, (ii) then – for each pair of D - all the pairs of DP and (iii) finally the DP shared by each pair of DP (see main text for ‘shared DP’). However, this procedure is computationally burdensome. Thus, in Module 4, we follow a bottom-top approach (starting from the DP and not from the D), which requires retrieving all the hpo pairs and their common ancestors. The input of this module is the HPO ontology (as hpoid.txt). The output is **hpo\_pairs.txt**. Later on, in Module 6, we assemble - around the hpo-pairs of interest - the hpo-associated D and the D-associated PS. \*/

**Module 4.1 Retrieve the ontology from HPO**

/\* From <http://www.obofoundry.org/ontology/hp.html>, retrieve the HPO ontology (hpoid.txt). \*/

.import c:/SQLITE/LHPS/DO\_HPO/hpoid.txt tmp\_hpoid

/\* Retrieve the essential rows, i.e., row\_id, term\_id, term\_name and the parent term(s) id. \*/

CREATE TABLE **tmp1**(row\_id INT, term\_id TEXT, description TEXT);

INSERT INTO tmp1

SELECT DISTINCT \* FROM(

SELECT rowid, term\_id, description FROM tmp\_hpoid

WHERE description LIKE ‘id: HP:%’ OR description LIKE ‘name: %’ OR description LIKE ‘is\_a: HP:%’);

/\* Pair each hpo\_id (tmp1 as i) with its hpo\_name (tmp1 as j) and parent id (tmp1 as k), by self-joining tmp1 twice. \*/

CREATE TABLE **tmp2**(hpo\_id TEXT, hpo\_name TEXT, hpo\_parent\_id TEXT);

INSERT INTO tmp2

SELECT DISTINCT \* FROM(

SELECT i.description AS hpo\_id, j.description AS hpo\_name, k.description AS hpo\_parent\_id

FROM tmp1 i

INNER JOIN tmp1 j ON (i.term\_id = j.term\_id) AND (i.description LIKE ‘id: HP:%’) AND (j.description LIKE ‘name: %’)

INNER JOIN tmp1 k ON (i.term\_id = k.term\_id) AND (i.description LIKE ‘id: HP:%’) AND (k.description LIKE ‘is\_a: HP:%’));

/\* Make some changes in style (e.g., ‘id: HP:XXXXXXX’ becomes simply ‘HP\_XXXXXXX’). \*/

UPDATE tmp2 SET hpo\_id = REPLACE (hpo\_id,’id: HP:’,’HP\_’);

UPDATE tmp2 SET hpo\_name = REPLACE (hpo\_name,’name: ’,’’);

UPDATE tmp2 SET hpo\_parent\_id = REPLACE (hpo\_parent\_id,‘is\_a: HP:’,’HP\_’);

/\* For the parent terms, divide the id and the name in two distinct columns. \*/

ALTER TABLE tmp2 ADD COLUMN hpo\_parent\_id\_def TEXT;

UPDATE tmp2 SET hpo\_parent\_id\_def = SUBSTR(hpo\_parent\_id,1,10);

ALTER TABLE tmp2 ADD COLUMN hpo\_parent\_name TEXT;

UPDATE tmp2 SET hpo\_parent\_name = SUBSTR(hpo\_parent\_id,14,LENGTH(hpo\_parent\_id));

/\* Assemble table hpo\_ontology with all the essential information from the ontology. \*/

CREATE TABLE **hpo\_ontology**(rowid INTEGER PRIMARY KEY, hpo\_id TEXT, hpo\_name TEXT, hpo\_parent\_id TEXT, hpo\_parent\_name TEXT);

INSERT INTO hpo\_ontology(hpo\_id, hpo\_name, hpo\_parent\_id, hpo\_parent\_name)

SELECT DISTINCT \* FROM(

SELECT hpo\_id, hpo\_name, hpo\_parent\_id\_def, hpo\_parent\_name

FROM tmp2 ORDER BY hpo\_id ASC);

**Module 4.2 Calculate the distance from the root in HPO**

/\* From hpo\_ontology, retrieve the *distance from the root* (‘HP\_0000001’,‘All’) by a *depth-first* search. The distance is not essential, but it is useful (further down the road) to discard all the hpo\_i/hpo\_j pairs whose common ancestor has a distance from the root < threshold (e.g < 2). \*/

/\* This is hpo\_ontology arranged hierarchically from the root to the leaves. \*/

CREATE TABLE **hpo\_ontology\_level**(hpo\_id TEXT, hpo\_name TEXT, dist\_root INT);

INSERT INTO hpo\_ontology\_level

SELECT \* FROM(

WITH RECURSIVE distance(hpo\_id, hpo\_name, level) AS (

VALUES (‘HP\_0000001’,‘All’,0)

UNION ALL

SELECT hpo\_ontology.hpo\_id, hpo\_ontology.hpo\_name, distance.level+1

FROM hpo\_ontology

JOIN distance ON hpo\_ontology.hpo\_parent\_id = distance.hpo\_id

ORDER BY distance.level+1 **DESC**

)

SELECT hpo\_id, SUBSTR(‘…………………………’,1,level+2)||hpo\_name, level FROM distance

);

**Module 4.3 Retrieve the ancestors of each term in HPO**

/\* Note that hpo\_ontology is not ordered, that is, each row simply states that a given term (hpo\_id) is son of another term (hpo\_parent\_id). Thus, we need to order the terms, in such a way that each term (query\_term\_id) is followed by all its ancestors - up to the root (‘HP\_0000001’,‘All’) – ordered in increasing distance from the query (or, in other words, in decreasing distance from the root). To this aim, we perform a nested recursive (*depth-first*) search of hpo\_ontology. Note also that ‘DESC’ ensures that the search is *depth-first*, whereby the lowest levels terms (the ones closest to the query) are processed first. For a *breadth-first* search (the highest levels being processed first), omit ‘DESC’. \*/

/\* Nested recursive (*depth-first*) search of the ancestors of a query down to the root. \*/

CREATE TABLE **tmp\_hpo\_ancestors**(progressive INT, hpo\_id TEXT, hpo\_name TEXT, level INT);

INSERT INTO tmp\_hpo\_ancestors

SELECT DISTINCT \* FROM(

WITH RECURSIVE *ancestors*(PROGR, hpo\_id, hpo\_name, DEPTH) AS (

SELECT rowid, hpo\_id, hpo\_name, 0 FROM hpo\_ontology WHERE rowid IN

(

WITH RECURSIVE proband(rowid) AS (

SELECT rowid FROM hpo\_ontology WHERE rowid = 1

UNION ALL

SELECT rowid+1 FROM hpo\_ontology

WHERE rowid < 17164

)

SELECT \* FROM proband

)

**UNION ALL**

SELECT ancestors.PROGR, hpo\_ontology.hpo\_parent\_id, hpo\_ontology.hpo\_parent\_name, DEPTH+1

FROM hpo\_ontology, ancestors

WHERE hpo\_ontology.hpo\_id = ancestors.hpo\_id

AND hpo\_ontology.hpo\_parent\_id IS NOT NULL

)

SELECT \* FROM ancestors ORDER BY PROGR, DEPTH);

/\* Insert the id and name of the query terms (from hpo\_ontology). \*/

CREATE TABLE **hpo\_ancestors**(query\_term\_id TEXT, hpo\_id TEXT, hpo\_name TEXT, dist\_query INT, dist\_root INT);

INSERT INTO hpo\_ancestors

SELECT DISTINCT \* FROM(

SELECT l.hpo\_id AS query\_term\_id, r1.hpo\_id AS hpo\_id, r1.hpo\_name, r1.level AS dist\_query, r2.dist\_root

FROM hpo\_ontology l

JOIN tmp\_hpo\_ancestors r1 ON l.rowid = r1.progressive

JOIN hpo\_ontology\_level r2 ON r1.hpo\_id = r2.hpo\_id

) ORDER BY query\_term\_id, dist\_query;

.once c:/SQLITE/LHPS/outputs/hpo\_ancestors.txt

SELECT \* FROM hpo\_ancestors;

DROP TABLE tmp\_hpoid; DROP TABLE tmp1; DROP TABLE tmp2; DROP TABLE tmp\_hpo\_ancestors;

**Module 4.4 Pairing the DP (of Module 2) to find the shared DP term(s)**

/\* Retrieve all the ancestors of each possible DP pair (from HPO) and identify (in **hpo\_comm\_anc**) the common ancestor for each pair. However, as creating hpo\_comm\_anc directly from hpo\_ancestors is computationally heavy, restrict the search (in tmp2) to the hpo\_id of interest (i.e., the hpo\_id in ps2mim2hpo, which are listed in tmp1). \*/

CREATE TABLE **tmp1**(hpo\_id TEXT);

INSERT INTO tmp1 SELECT DISTINCT(hpo\_id) FROM ps2mim2hpo ORDER BY hpo\_id ASC;

/\* tmp2 is a subset of hpo\_ancestors (restricted to the hpo\_id of interest). \*/

CREATE TABLE **tmp2**(query\_term\_id TEXT, hpo\_id TEXT, hpo\_name TEXT, dist\_query INT, dist\_root INT);

INSERT INTO tmp2

SELECT DISTINCT \* FROM(

SELECT l.hpo\_id AS query\_term\_id, r.hpo\_id, r.hpo\_name, r.dist\_query, r.dist\_root

FROM tmp1 l

JOIN hpo\_ancestors r ON l.hpo\_id = r.query\_term\_id

) ORDER BY query\_term\_id ASC, dist\_query ASC;

/\* Self-join tmp2. \*/

CREATE TABLE **hpo\_comm\_anc**(hpo\_id\_i TEXT, hpo\_id\_j TEXT, common\_id TEXT, common\_name TEXT, common\_dist\_root INT, from\_query\_i REAL, from\_query\_j REAL);

INSERT INTO hpo\_comm\_anc

SELECT hpo\_id\_i, hpo\_id\_j, common\_id, common\_name, common\_dist\_root, from\_query\_i, from\_query\_j FROM(

SELECT i.query\_term\_id AS hpo\_id\_i, j.query\_term\_id AS hpo\_id\_j,

i.hpo\_id AS common\_id, i.hpo\_name AS common\_name,

i.dist\_root AS common\_dist\_root,

i.dist\_query AS from\_query\_i, j.dist\_query AS from\_query\_j

FROM tmp2 i

INNER JOIN tmp2 j ON (i.hpo\_id = j.hpo\_id) AND (i.query\_term\_id != j.query\_term\_id) AND (i.dist\_root > 2)

) ORDER BY hpo\_id\_i, hpo\_id\_j;

ALTER TABLE hpo\_comm\_anc ADD COLUMN avg\_from\_query REAL;

UPDATE hpo\_comm\_anc SET avg\_from\_query = (from\_query\_i+from\_query\_j)/2;

DROP TABLE tmp1; DROP TABLE tmp2;

/\* Create a table containing all the hpo-hpo pairs (hpo\_i and hpo\_j, with j ≠ j) according to the two following rules (see manuscript; **Rule 1** is simply hpo\_i = hpo\_j and is omitted here).

**Rule 2**, hpo\_i ≠ hpo\_j, but they have the same father, i.e., the father of hpo\_i is the father of hpo\_j (i.e., avg\_from\_query = 1.0 AND from\_query\_i = from\_query\_j = 1.0.

**Rule 3**, hpo\_i ≠ hpo\_j, but hpo\_j is the father of hpo\_i (or hpo\_i is the father of hpo\_j) i.e., avg\_from\_query = 0.5. \*/

CREATE TABLE **hpo\_pairs**(hpo\_id\_i TEXT, hpo\_id\_j TEXT, common\_id TEXT, from\_query\_i INT, from\_query\_j INT, avg\_from\_query REAL, rule INT);

/\* Rule 2 \*/

INSERT INTO hpo\_pairs

SELECT DISTINCT \* FROM(

SELECT hpo\_id\_i, hpo\_id\_j, common\_id, from\_query\_i, from\_query\_j, avg\_from\_query, 2

FROM hpo\_comm\_anc

WHERE (avg\_from\_query = 1.0) AND (from\_query\_i = from\_query\_j));

/\* Rule 3 \*/

INSERT INTO hpo\_pairs

SELECT DISTINCT \* FROM(

SELECT hpo\_id\_i, hpo\_id\_j, common\_id, from\_query\_i, from\_query\_j, avg\_from\_query, 3

FROM hpo\_comm\_anc

WHERE avg\_from\_query = 0.5);

.once c:/SQLITE/LHPS/outputs/hpo\_pairs.txt

SELECT DISTINCT \* FROM hpo\_pairs;