**SCRIPT SQL (Inputs and Outputs)**



**Module 1. Retrieving the Diseases (D) and the Phenotypic Series (PS) from OMIM**

/\* Retrieve the D and their grouping into PS from the most recent **morbidmap.txt** and **phenotypicSeries.txt** files othe OMIM website. \*/

.open c:/sqlite/LHPS/lhps.db

.mode tabs

.headers on

/\* Import **morbidmap.txt** and select the columns of interest (i.e., D as phenotype name and gene as MIM number) into table tmp2\_morbidmap. \*/

.import c:/sqlite/LHPS/inputs/**morbidmap.txt** tmp1\_morbidmap

CREATE TABLE **tmp2\_morbidmap**(mim\_dgp\_id INT, mim\_name TEXT);

INSERT INTO tmp2\_morbidmap

SELECT DISTINCT \* FROM(SELECT MIM\_number, Phenotype FROM tmp1\_morbidmap);

/\* From each phenotype (i.e., D as mim\_name), extract the leftmost character and the key, to retain only the real D (i.e., first character in string is different from ‘?’, ‘[‘ or ‘{‘) and the key is 3 because the molecular phenotype is known). \*/

ALTER TABLE tmp2\_morbidmap ADD COLUMN start TEXT;

UPDATE tmp2\_morbidmap SET start = SUBSTR(mim\_name, 1,1);

ALTER TABLE tmp2\_morbidmap ADD COLUMN key TEXT;

UPDATE tmp2\_morbidmap SET key = SUBSTR(mim\_name, -3,3);

/\* Delete the following disease phenotypes: (i) susceptibility to multi-factorial disorders (start = ‘{‘); (ii) non-diseases (start = ‘[‘); (iii) unconfirmed diseases (start = ‘?‘). \*/

DELETE FROM tmp2\_morbidmap WHERE start = ‘{’;

DELETE FROM tmp2\_morbidmap WHERE start = ‘[’;

DELETE FROM tmp2\_morbidmap WHERE start = ‘?’;

/\* Delete the following D: (i) D with unknown gene defect (key = ‘(1)‘); (ii) D with unknown mutation (key = ‘(2)‘); (iii) D caused by multiple genes (key = ‘(4)‘). This way, we retain only D with known molecular basis (key = 3). \*/

DELETE FROM tmp2\_morbidmap WHERE key = ‘(1)’;

DELETE FROM tmp2\_morbidmap WHERE key = ‘(2)’;

DELETE FROM tmp2\_morbidmap WHERE key = ‘(4)’;

/\* As the Disease Gene Products (DGP) are designated in OMIM with MIM identifiers, retrieve the DGP as entrez(NCBI)/symbol(HUGO). Then, to convert the DGP from OMIM id (‘mim\_dgp\_id’) to Entrez id (entrez\_id), first import the mim2gene file from OMIM. \*/

.import c:/sqlite/LHPS/inputs/**mim2gene.txt** tmp\_mim2gene

CREATE TABLE **mim2gene**(dgp\_mim INT, type TEXT, entrez\_id INT, symbol TEXT);

INSERT INTO mim2gene

SELECT DISTINCT \* FROM(SELECT Mim\_Number, Type, Entrez, Symbol FROM tmp\_mim2gene);

/\* From mim2gene, delete the rows in which the D has been either moved or removed. \*/

DELETE FROM mim2gene WHERE type = ‘moved/removed’;

/\*.Join the D (from tmp2\_morbidmap) with the DGP (from mim2gene) to retrieve the D that have an entrez\_id. \*/

CREATE TABLE **tmp3\_morbidmap**(mim\_dgp\_id INT, mim\_name TEXT, mim\_type TEXT, entrez\_id INT);

INSERT INTO tmp3\_morbidmap

SELECT DISTINCT \* FROM(

SELECT l.mim\_dgp\_id, l.mim\_name, r.type, r.entrez\_id

FROM tmp2\_morbidmap l

LEFT JOIN mim2gene r

ON l.mim\_dgp\_id = r.dgp\_mim

);

/\* Retain the DGP that are protein-coding genes. Join the D (from tmp3\_morbidmap) with the symbols (and type of gene) from the (ftp version of) Entrez Gene, limited to *H. sapiens* (taxon 9606). \*/

.import c:/sqlite/LHPS/inputs/**Homo\_sapiens.gene\_info.txt** tmp\_hs

DELETE FROM tmp\_hs WHERE tax\_id != ‘9606’; /\* Retain H. sapiens only. \*/

CREATE TABLE **hs\_entrez**(entrez\_id INT, symbol TEXT, dbXrefs TEXT, gene\_type TEXT);

INSERT INTO hs\_entrez

SELECT DISTINCT \* FROM(

SELECT GeneID, Symbol, SUBSTR(dbXrefs,1,10), type\_of\_gene FROM tmp\_hs);

/\* Retain only the xrefs to OMIM, in format ‘MIM:XXXXXX’. \*/

UPDATE hs\_entrez SET dbXrefs = NULL WHERE SUBSTR(dbXrefs,1,3) <> ‘MIM’;

/\* Convert the xrefs format ‘MIM:XXXXXX’ into ‘XXXXXX’. \*/

UPDATE hs\_entrez SET dbXrefs =

REPLACE(dbXrefs,dbXrefs,SUBSTR(dbXrefs,5,10)) WHERE dbXrefs IS NOT NULL;

/\* As mentioned above, join the D (from tmp3\_morbidmap) with the symbols (from hs\_entrez). \*/

CREATE TABLE **tmp4\_morbidmap**(mim\_name TEXT, dgp\_mim INT, entrez\_id INT, symbol TEXT, gene\_type TEXT, dbXrefs INT);

INSERT INTO tmp4\_morbidmap

SELECT DISTINCT \* FROM(

SELECT l.mim\_name, l.mim\_dgp\_id, l.entrez\_id, r.symbol, r.gene\_type, r.dbXrefs

FROM tmp3\_morbidmap l

LEFT JOIN hs\_entrez r

ON l.entrez\_id = r.entrez\_id

);

/\* Delete the rows in which the gene is NOT a protein-coding gene (e.g., ncRNA, other, snRNA, snoRNA, pseudo, unknown). Also, delete the few D without a corresponding gene (e.g., ‘Myelodysplasia syndrome-1’ 600049, ‘Mulchandani-Bhoj-Conlin syndrome-1’ 617352 and ‘Histiocytosis-lymphadenopathy plus syndrome’ 602782). Note that there might be a discrepancy in the DGP count as dgp\_mim v. entrez\_id/symbol, as one gene (6473 in entrez) corresponds to two items in OMIM, mim 312865 (SHOX) and mim 400020 (SHOY). \*/

DELETE FROM tmp4\_morbidmap WHERE gene\_type != ‘protein-coding’;

DELETE FROM tmp4\_morbidmap WHERE entrez\_id = ‘’;

/\* Retrieve the phenotype (D) id, where available. Check and make correction (e.g., 167400). Then, delete the phenotypes in which the rightmost characters are not 6-digit id. \*/

UPDATE tmp4\_morbidmap SET mim\_name = ‘Paroxysmal extreme pain disorder, 167400 (3)’ WHERE mim\_name = ‘Paroxysmal extreme pain disorder, 167400, (3)’;

/\* Then, remove the rightmost substring “ (3)”. \*/

UPDATE tmp4\_morbidmap SET mim\_name = SUBSTR(mim\_name,1,LENGTH(mim\_name)-4);

/\* Retrieve the rightmost substring of the mim\_name, which contains the phen\_id. Where a phen-id is available, the outcome is a 6-digit integer. Otherwise, the outcome is a non-integer value that will be filtered out in omim-def when setting phen\_id < = 999999, i.e., a non-numeric value). \*/

ALTER TABLE tmp4\_morbidmap ADD COLUMN phen\_id INT;

UPDATE tmp4\_morbidmap SET phen\_id = SUBSTR(mim\_name,-6,6);

UPDATE tmp4\_morbidmap SET phen\_id = NULL WHERE phen\_id > 999999;

/\* Table **omim\_def** is the final version of the morbid map comprehensive of the DGP. \*/

CREATE TABLE **omim\_def**(phen\_mim INT, mim\_name TEXT, dgp\_mim INT, entrez\_id INT, symbol TEXT);

INSERT INTO omim\_def

SELECT DISTINCT \* FROM(

SELECT phen\_id, mim\_name, dgp\_mim, entrez\_id, symbol FROM tmp4\_morbidmap

WHERE phen\_id IS NOT NULL

ORDER BY phen\_id ASC, mim\_name ASC

);

UPDATE omim\_def SET mim\_name = SUBSTR(mim\_name,1,LENGTH(mim\_name)-8);

.once c:/sqlite/LHPS/outputs/**omim\_def.txt**

SELECT \* FROM omim\_def;

DROP TABLE tmp1\_morbidmap; DROP TABLE tmp2\_morbidmap;

DROP TABLE tmp3\_morbidmap; DROP TABLE tmp4\_morbidmap;

DROP TABLE tmp\_mim2gene; DROP TABLE tmp\_hs; DROP TABLE mim2gene;

DROP TABLE hs\_entrez;

/\* Import the PS list as phenotypicSeries.txt (available from OMIM upon request). \*/

.import c:/sqlite/LHPS/inputs/**phenotypicSeries.txt** tmp\_ps

/\* To extract the PS titles from tmp\_ps, retain the rows where column mim\_name - the name of the PS – is empty (‘’), because the PS name is reported in column mim\_id). \*/

CREATE TABLE **ps\_names**(ps\_id TEXT, ps\_name TEXT);

INSERT INTO ps\_names

SELECT DISTINCT \* FROM(SELECT ps\_id, mim\_id FROM tmp\_ps WHERE mim\_name = ‘’);

/\* Remove from tmp\_ps the rows containing just the PS title (as explained above) and not a D within the PS. \*/

DELETE FROM tmp\_ps WHERE mim\_name = ‘’;

CREATE TABLE **ps**(ps\_id TEXT, mim\_id INT, mim\_name TEXT);

INSERT INTO ps

SELECT DISTINCT \* FROM(SELECT \* FROM tmp\_ps)

ORDER BY ps\_id ASC, mim\_id ASC, mim\_name ASC;

/\* In ps\_dictionary, join the two inputs of OMIM (the morbid map as ‘omim\_def’ and the PS table as ‘ps\_names’). \*/

CREATE TABLE **ps\_dictionary**(ps\_id TEXT, ps\_name TEXT, mim\_id INT, mim\_name\_morbidmap TEXT, dgp\_mim INT, entrez INT, symbol TEXT);

INSERT INTO ps\_dictionary

SELECT DISTINCT \* FROM(

SELECT l.ps\_id, r1.ps\_name, l.mim\_id,

r2.mim\_name, r2.dgp\_mim, r2.entrez\_id, r2.symbol

FROM ps l

LEFT JOIN ps\_names r1 ON l.ps\_id = r1.ps\_id

LEFT JOIN omim\_def r2 ON l.mim\_id = r2.phen\_mim

);

/\* The ps\_dictionary table, contains many D that have not been characterized in molecular terms. For instance, many D have been assigned to a certain PS but still have no DGP associated (and therefore have a key different from 3). So, delete these D. \*/

DELETE FROM ps\_dictionary WHERE dgp\_mim IS NULL;

.once c:/sqlite/LHPS/outputs/**ps\_dictionary.txt**

SELECT \* FROM ps\_dictionary;

DROP TABLE tmp\_ps; DROP TABLE ps\_names;

**Module 2. Retrieving the disease phenotypes (DP) from HPO**

/\* First, merge the OMIM-encoded D with the HPO-encoded DP. Assemble the *2*-tuples composed of a D (as MIM id) and the D-associated DP (as HPO id). Only DP that are descendants of HP:0000118 (Phenotypic abnormality) are used, while all the descendants of the other four children of HP:0000001 (All), i.e., HP:0000005 (Mode of inheritance), HP:0012823 (Clinical modifier), HP:0040006 (Mortality/Aging) and HP:0040279 (Frequency), have been discarded.

The annotation data have been obtained from the HPO at the following web site: <http://compbio.charite.de/jenkins/job/hpo.annotations/lastStableBuild/artifact/misc/phenotype_annotation.tab> and then slightly modified (as mim2hpo.txt). \*/

.import c:/sqlite/LHPS/inputs/**mim2hpo.txt** tmp\_mim2hpo

CREATE TABLE **mim2hpo**(mim\_id INT, hpo\_id TEXT, entrez INT);

INSERT INTO mim2hpo

SELECT DISTINCT \* FROM(SELECT mim\_id, hpo\_id, entrez FROM tmp\_mim2hpo);

/\* Assemble the 2-tuples composed of a DP and its associated Informational Content (IC). Data are from file IC\_HPO.txt (from A. Gamba) and have been sligthly modified as hpo2ic.txt. The IC values are normalized in the range [0.0, 1.0]. \*/

.import c:/sqlite/LHPS/inputs/**hpo2ic.txt** tmp\_hpo2ic

CREATE TABLE **hpo2ic**(hpo\_id TEXT, ic REAL);

INSERT INTO hpo2ic SELECT DISTINCT \* FROM(SELECT hpo\_id, ic FROM tmp\_hpo2ic);

/\* Merge the two tables above (mim2hpo and hpo2ic) into the **mim2hpo2ic** table. Also, add the names of the phenotypes (from the hpo\_dictionary, originally hpo\_names.txt). The HPO dictionary was obtained by elaborating the HPO ontology (<http://www.obofoundry.org/ontology/hp.html>) as hpo.obo file. \*/

.import c:/sqlite/LHPS/inputs/**hpo\_dictionary.txt** hpo\_dictionary

CREATE TABLE **mim2hpo2ic**(mim\_id INT, hpo\_id TEXT, ic REAL, hpo\_name TEXT);

INSERT INTO mim2hpo2ic

SELECT DISTINCT \* FROM(

SELECT l.mim\_id, l.hpo\_id, ROUND(r1.ic,3), r2.hpo\_name

FROM mim2hpo l

LEFT JOIN hpo2ic r1 ON l.hpo\_id = r1.hpo\_id

LEFT JOIN hpo\_dictionary r2 ON l.hpo\_id = r2.hpo\_id

);

/\* Second, merge the DP annotations of the D with the PS annotations of the D. Before that, generate the *2*-tuples composed of a PS (as PS id) and the PS-associated D (as MIM id). \*/

CREATE TABLE **ps2mim**(ps\_id TEXT, ps\_name TEXT, mim\_id INT, mim\_name TEXT, entrez INT, symbol TEXT);

INSERT INTO ps2mim

SELECT DISTINCT \* FROM(

SELECT ps\_id, ps\_name, mim\_id, mim\_name\_morbidmap, entrez, symbol FROM ps\_dictionary

);

/\*Finally, assemble the 3-tuples composed of a PS, a D and a DP (with its IC). \*/

CREATE TABLE **ps2mim2hpo**(ps\_id TEXT, mim\_id INT, hpo\_id TEXT, ic REAL, ps\_name TEXT, hpo\_name TEXT);

INSERT INTO ps2mim2hpo

SELECT DISTINCT \* FROM(

SELECT l.ps\_id AS ps\_id, l.mim\_id AS mim\_id, r.hpo\_id, r.ic AS ic, l.ps\_name, r.hpo\_name

FROM ps2mim l

LEFT JOIN mim2hpo2ic r ON l.mim\_id = r.mim\_id

)

ORDER BY ps\_id ASC, mim\_id ASC, ic DESC;

/\* Delete the rows lacking HPO annotations of the D. \*/

DELETE FROM ps2mim2hpo WHERE hpo\_id IS NULL;

.once c:/SQLITE/LHPS/outputs/**ps2mim2hpo.txt**

SELECT \* FROM ps2mim2hpo;

DROP TABLE tmp\_mim2hpo; DROP TABLE tmp\_hpo2ic;

**Module 3. Assembling the bipartite D-DP graph**

/\* The most obvious way for assembling the D-DP bipartite graph consists of simply retrieving the the D-DP *2*-tuples from ps2mim2hpo. However, as in ps2mim2hpo some mim D belong to more PS, for ease of analysis, we discard these pluri\_PS D. \*/

/\* First, identify the D that belong to more PS. \*/

CREATE TABLE tmp1(mim\_id INT, ps4mim INT);

INSERT INTO tmp1

SELECT DISTINCT \* FROM(

SELECT mim\_id, COUNT(DISTINCT ps\_id) AS ps4mim FROM ps2mim2hpo

GROUP BY mim\_id ORDER BY ps4mim DESC, mim\_id ASC)

WHERE ps4mim > 1;

/\* Then, with this information, assemble the bipartite D-DP graph. \*/

CREATE TABLE **bipartite**(ps\_id TEXT, mim\_id INT, hpo\_id TEXT, ic REAL, ps\_name TEXT, hpo\_name TEXT, pluri\_ps INT);

INSERT INTO bipartite

SELECT DISTINCT \* FROM(

SELECT l.\*, r.mim\_id FROM ps2mim2hpo l

LEFT JOIN tmp1 r ON (l.mim\_id = r.mim\_id)

);

/\* Delete the rows of bipartite that contain the D with multiple PS annotations. \*/

UPDATE bipartite SET pluri\_ps = 1 WHERE pluri\_ps IS NOT NULL;

UPDATE bipartite SET pluri\_ps = 0 WHERE pluri\_ps IS NULL;

DELETE FROM bipartite WHERE pluri\_ps = 1;

.once c:/SQLITE/LHPS/outputs/**bipartite.txt**

SELECT DISTINCT \* FROM(SELECT mim\_id, hpo\_id, ic FROM bipartite);

/\* This is NOT the definitive bipartite D\_DP. See below Module 6. \*/

DROP TABLE tmp1;

**Module 4. Pairing the DP pairs from HPO and finding 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;

**Module 5. Retrieving the etiological annotations from DO**

/\*Disease Ontology (DO) is an ontology for the etiological classification of human D. We use DO to map each OMIM-encoded D to an etiological class. Although detailed in its structure (with up to a dozen of generations for some D), we need DO just to classify each D into the major branches of the DO tree. We focus on the eight children of the common root DO:0000004 ‘disease’, i.e., ‘syndrome’, ‘genetic disease’, ‘physical disorder’, ‘disease by infectious agent’, ‘disease of metabolism’, ‘disease of mental health’, ‘disease of cellular proliferation’ and ‘disease of anatomical entity’. In addition, we also use the twelve child terms of DO:0000007 ‘disease of anatomical entity’, i.e., cardiovascular, endocrine, gastrointestinal, hematopoietic, immune, integumentary, musculoskeletal, nervous, reproductive, respiratory, thoracic and urinary system disease. Note that, besides DO, also Orphanet describes the D etiologically. Unfortunately, however, all the mappings from Orphanet to DO are just DO:630 ‘genetic disease’ \*/

**5.1 Mapping OMIM diseases to the corresponding DO term(s)**

/\* From [Human Disease Ontology (obofoundry.org)](http://www.obofoundry.org/ontology/doid.html), retrieve the DO ontology (doid.txt). \*/

.import c:/SQLITE/LHPS/DO\_HPO/doid.txt tmp\_doid

/\* Retrieve the essential rows, i.e., term\_id, term\_name, external reference to OMIM 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\_doid

WHERE description LIKE ‘id: DOID:%’ OR description LIKE ‘name: %’ OR description LIKE ‘xref: OMIM:%’ OR description LIKE ‘is\_a: DOID:%’

);

/\* Pair each do\_id (tmp1 as i) with its do\_name (tmp1 as j), parent id (tmp1 as k) and (where available) its xref to OMIM (tmp1 as m), by self-joining tmp1 three times. \*/

CREATE TABLE **tmp2**(do\_id TEXT, do\_name TEXT, do\_parent\_id TEXT, omim\_id INT);

INSERT INTO tmp2

SELECT DISTINCT \* FROM(

SELECT i.description AS do\_id, j.description AS do\_name, k.description AS do\_parent\_id, m.description AS omim\_id

FROM tmp1 i

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

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

LEFT JOIN tmp1 m ON (i.term\_id = m.term\_id) AND (i.description LIKE ‘id: DOID:%’) AND (m.description LIKE ‘xref: OMIM:%’)

);

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

UPDATE tmp2 SET do\_id = REPLACE (do\_id,’id: DOID:’,’DO\_’);

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

UPDATE tmp2 SET do\_parent\_id = REPLACE (do\_parent\_id,‘is\_a: DOID:’,’DO\_’);

UPDATE tmp2 SET omim\_id = REPLACE (omim\_id,‘xref: OMIM:’,’’);

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

/\* Replace the substring ‘ ! ‘ with ‘!’. \*/

UPDATE tmp2 SET do\_parent\_id = REPLACE(do\_parent\_id,’ ! ‘,’!’);

/\* Add a new column for the name of the parent term (after the ‘!’) \*/

ALTER TABLE tmp2 ADD COLUMN do\_parent\_name; UPDATE tmp2 SET do\_parent\_name = SUBSTR(do\_parent\_id,INSTR(do\_parent\_id,’!’)+1,LENGTH(do\_parent\_id));

/\* Update the column for the id of the parent term (before the ‘!’). \*/

UPDATE tmp2 SET do\_parent\_id = SUBSTR(do\_parent\_id,1,INSTR(do\_parent\_id,’!’)-1);

/\* Also, as the DO terms are annotated in two different ways (e.g., DO\_7 OR DO\_0000007), convert tmp2 into the correct format (both columns do\_id and do\_parent\_id). \*/

/\* For the do\_id. \*/

ALTER TABLE tmp2 ADD COLUMN do\_id\_int INT;

UPDATE tmp2 SET do\_id\_int = REPLACE(do\_id,’DO\_’,’’);

ALTER TABLE tmp2 ADD COLUMN do\_id\_def TEXT;

UPDATE tmp2 SET do\_id\_def = SUBSTR(“DO\_0000000”,1,10-LENGTH(do\_id\_int))||do\_id\_int;

/\* For the do\_parent\_id. \*/

ALTER TABLE tmp2 ADD COLUMN do\_parent\_id\_int INT;

UPDATE tmp2 SET do\_parent\_id\_int = REPLACE(do\_parent\_id,’DO\_’,’’);

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

UPDATE tmp2 SET do\_parent\_id\_def = SUBSTR(“DO\_0000000”,1,10-LENGTH(do\_parent\_id\_int))||do\_parent\_id\_int;

/\* Assemble table do\_ontology with all the essential information from the ontology. Also, as some DO2MIM mappings are not with a mim\_id but with a PS, convert (by left joining tmp2 with the ps\_dictonary) the ps\_id (column tmp\_omim\_id) into the corresponding mim\_ids (column omim\_id). \*/

CREATE TABLE **do\_ontology**(rowid INTEGER PRIMARY KEY, do\_id TEXT, do\_name TEXT, do\_parent\_id TEXT, do\_parent\_name TEXT, tmp\_omim\_id INT, omim\_id INT);

INSERT INTO do\_ontology(do\_id, do\_name, do\_parent\_id, do\_parent\_name, tmp\_omim\_id, omim\_id)

SELECT DISTINCT \* FROM(

SELECT l.do\_id\_def AS do\_id, l.do\_name, l.do\_parent\_id\_def AS do\_parent\_id, l.do\_parent\_name, l.omim\_id AS tmp\_omim\_id, r.mim\_id AS omim\_id

FROM tmp2 l

LEFT JOIN ps\_dictionary r ON l.omim\_id = r.ps\_id

) ORDER BY do\_id ASC;

/\* Fill in the missing cells in omim\_id. \*/

UPDATE do\_ontology SET omim\_id =

(CASE WHEN(omim\_id IS NULL) THEN tmp\_omim\_id ELSE omim\_id END);

**Module 5.2 Calculate the distance from the root in DO**

/\* From do\_ontology, retrieve the *distance from the root* (‘DO\_0000004’,‘disease’) 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 do\_ontology arranged hierarchically from the root to the leaves. \*/

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

INSERT INTO do\_ontology\_level

SELECT \* FROM(

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

VALUES (‘DO\_0000004’,‘disease’,0)

UNION ALL

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

FROM do\_ontology

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

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

)

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

);

**Module 5.3 Retrieve the ancestors of each term in DO**

/\* As it was done with HPO, we we perform a nested recursive (*depth-first*) search of do\_ontology. \*/

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

CREATE TABLE tmp\_do\_ancestors(progressive INT, do\_id TEXT, do\_name TEXT, level INT);

INSERT INTO **tmp\_do\_ancestors**

SELECT DISTINCT \* FROM(

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

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

(

WITH RECURSIVE proband(rowid) AS (

SELECT rowid FROM do\_ontology WHERE rowid = 1

UNION ALL

SELECT rowid+1 FROM do\_ontology

WHERE rowid < 10920

)

SELECT \* FROM proband

)

**UNION ALL**

SELECT ancestors.PROGR, do\_ontology.do\_parent\_id,

do\_ontology.do\_parent\_name, DEPTH+1

FROM do\_ontology, ancestors

WHERE do\_ontology.do\_name = ancestors.do\_name

AND do\_ontology.do\_parent\_name IS NOT NULL

)

SELECT \* FROM ancestors ORDER BY PROGR, DEPTH

);

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

INSERT INTO do\_ancestors

SELECT DISTINCT \* FROM(

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

FROM do\_ontology l

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

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

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

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

SELECT \* FROM do\_ancestors;

DROP TABLE tmp1; DROP TABLE tmp2;

DROP TABLE tmp\_doid; DROP TABLE tmp\_do\_ancestors;

**Module 5.4 Map each D to an ancestor term in DO (for the aetiological classification of the D)**

/\* In the **omim2do** table, retrieve (as do\_anc\_def) the *low-specificity* ancestor of the DO term that annotate the D in the omim\_def table (from Module 1). Specifically, retrieve the 8 children of the root DO:0000004 (Disease), i.e., those with dist\_root = 1 (including DO:0000007, Disease of anatomical entity), as well as the 12 children of DO:0000007), i.e., those with dist\_root = 2.\*/

CREATE TABLE **omim2do**(mim\_id INT, do\_id TEXT, do\_name TEXT, do\_anc2 TEXT, do\_anc2\_name TEXT, do\_anc1 TEXT, do\_anc1\_name TEXT);

INSERT INTO omim2do

SELECT DISTINCT \* FROM(

SELECT l.phen\_mim AS mim\_id,

r1.do\_id AS do\_id, r1.do\_name AS do\_name, /\* do\_id, the DO term annotating MIM \*/

r2.do\_id AS do\_anc2, r2.do\_name AS do\_anc2\_name, /\* the ancestor with d\_root = 2 \*/

r3.do\_id AS do\_anc1, r3.do\_name AS do\_anc1\_name /\* the ancestor with d\_root = 1 \*/

FROM omim\_def l

LEFT JOIN do\_ontology r1 ON l.phen\_mim = r1.omim\_id

LEFT JOIN do\_ancestors r2 ON (r1.do\_id = r2.query\_term\_id AND r2.dist\_root = 2)

LEFT JOIN do\_ancestors r3 ON (r1.do\_id = r3.query\_term\_id AND r3.dist\_root = 1)

);

/\* If anc1 is DO:0000007, then use its child as definitive annotation. \*/

ALTER TABLE omim2do ADD COLUMN do\_anc\_def; UPDATE omim2do SET do\_anc\_def =

(CASE WHEN do\_anc1 = ‘DO\_0000007’ THEN do\_anc2 ELSE do\_anc1 END);

ALTER TABLE omim2do ADD COLUMN do\_anc\_def\_name; UPDATE omim2do SET do\_anc\_def\_name =

(CASE WHEN do\_anc1 = ‘DO\_0000007’ THEN do\_anc2\_name ELSE do\_anc1\_name END);

/\* Add ‘DO\_0000000’ if a D has no mappings in DO. \*/

UPDATE omim2do SET do\_anc\_def = ‘DO\_0000000’ WHERE do\_id IS NULL;

UPDATE omim2do SET do\_anc\_def\_name = ‘no DO mapping’ WHERE do\_id IS NULL;

/\* For the analysis reported in the manuscript, whenever a mim\_id is annotated with more than one DO ancestor and one of these ancestors is DO:0000630 or ‘genetic diseases’, the latter is dropped, as in table **omim2do\_def**. Also, remove the D without DO mappings. \*/

CREATE TABLE **omim2do\_def**(mim\_id INT, do\_anc\_def TEXT, do\_anc\_def\_name TEXT);

INSERT INTO omim2do\_def

SELECT DISTINCT \* FROM(

SELECT mim\_id, do\_anc\_def, do\_anc\_def\_name

FROM omim2do WHERE mim\_id NOT IN(

SELECT mim\_id FROM(

SELECT mim\_id, COUNT(DISTINCT do\_anc\_def) AS do\_anc4mim

FROM omim2do GROUP BY mim\_id)

WHERE do\_anc4mim > 1 AND do\_anc\_def = ‘DO\_0000630’)

) WHERE do\_anc\_def != ‘DO\_0000000’;

.once c:/SQLITE/LHPS/outputs/omim2do\_def.txt

SELECT \* FROM omim2do\_def;

**5.5 Mapping the PS to a low-level DO term**

CREATE TABLE **ps2do**(ps\_id TEXT, ps\_name TEXT, mim\_id INT, do\_anc\_def TEXT, do\_anc\_def\_name TEXT);

INSERT INTO ps2do

SELECT DISTINCT \* FROM(

SELECT l.ps\_id, l.ps\_name, l.mim\_id,

r.do\_anc\_def, r.do\_anc\_def\_name

FROM ps\_dictionary l

LEFT JOIN omim2do\_def r ON (l.mim\_id = r.mim\_id)

) WHERE do\_anc\_def IS NOT NULL;

.once c:/SQLITE/LHPS/outputs/ps2do.txt

SELECT \* FROM ps2do;

**Module 6. Assembling the complete bipartite D-DP graph and the DDSN-C**

/\* The final aim is to assemble the definitive D-DP bipartite graph and the DDSN-C in which two D are joined (rule 1) if they share the same DP or (rule 2) if two of their DP share the same father or (rule 3) if one DP of the two D is the father of a DP in another D. As explained (in Module 4), we follow a bottom-top strategy starting from the hpo\_i/hpo\_j pairs (hpo\_pairs) up to the corresponding D (and finally to the corresponding PS). \*/

/\* First, by joining hpo\_pairs (from Module 4) with bipartite (from Module 3), assemble the D (and also the PS) around the heterologous hpo\_i/hpo\_j pairs (where hpo\_id\_i **≠** hpo\_id\_j). \*/

CREATE TABLE **hpo\_pairs\_hetero**(hpo\_id\_i TEXT, hpo\_id\_j TEXT, common\_hpo\_id TEXT, ps\_id\_i TEXT, mim\_id\_i INT, ps\_id\_j TEXT, mim\_id\_j INT);

INSERT INTO hpo\_pairs\_hetero

SELECT DISTINCT \* FROM(

SELECT l.hpo\_id\_i AS hpo\_id\_i, l.hpo\_id\_j AS hpo\_id\_j, l.common\_id AS common\_hpo\_id,

r1.ps\_id AS ps\_id\_i, r1.mim\_id AS mim\_id\_i,

r2.ps\_id AS ps\_id\_j, r2.mim\_id AS mim\_id\_j

FROM hpo\_pairs l

JOIN bipartite r1 ON l.hpo\_id\_i = r1.hpo\_id

JOIN bipartite r2 ON l.hpo\_id\_j = r2.hpo\_id

) WHERE mim\_id\_i != mim\_id\_j;

/\* Second, by self-joining bipartite (from Module 3), assemble the D (and also the PS) around the homologous hpo\_i/hpo\_j pairs (where hpo\_id\_i **=** hpo\_id\_j). \*/

CREATE TABLE **hpo\_pairs\_homo**(hpo\_id\_i TEXT, hpo\_id\_j TEXT, common\_hpo\_id TEXT, ps\_id\_i TEXT, mim\_id\_i INT, ps\_id\_j TEXT, mim\_id\_j INT);

INSERT INTO hpo\_pairs\_homo

SELECT DISTINCT \* FROM(

SELECT i.hpo\_id AS hpo\_id\_i, j.hpo\_id AS hpo\_id\_j, i.hpo\_id AS common\_id,

i.ps\_id AS ps\_id\_i, i.mim\_id AS mim\_id\_i,

j.ps\_id AS ps\_id\_j, j.mim\_id AS mim\_id\_j

FROM bipartite i

INNER JOIN bipartite j

ON (i.mim\_id < j.mim\_id) AND (i.hpo\_id = j.hpo\_id)

);

**6.2 The ‘redundant’ DDSN-C (more edges for the same node pair)**

/\* First, assemble all the (homologous and heterologous DP pairs) from above in table **tmp1\_ddsn**, which thus contains the 7-tuples [PSi>Di>DPi>DP(i,j)>DPj>Dj>PSj]. The resulting DDSN-C here is defined ‘redundant’, because there is often more than one edge (more than one shared DP) linking the same pair of D nodes Di and Dj. \*/

CREATE TABLE **tmp1\_ddsn**(ps\_id\_i TEXT, mim\_id\_i INT, hpo\_id\_i TEXT, common\_hpo\_id TEXT, hpo\_id\_j TEXT, mim\_id\_j INT, ps\_id\_j TEXT);

INSERT INTO tmp1\_ddsn

SELECT \* FROM(

SELECT ps\_id\_i, mim\_id\_i, hpo\_id\_i, common\_hpo\_id, hpo\_id\_j, mim\_id\_j, ps\_id\_j

FROM hpo\_pairs\_homo)

**UNION**

SELECT \* FROM(

SELECT ps\_id\_i, mim\_id\_i, hpo\_id\_i, common\_hpo\_id, hpo\_id\_j, mim\_id\_j, ps\_id\_j

FROM hpo\_pairs\_hetero WHERE mim\_id\_i < mim\_id\_j

**UNION**

SELECT ps\_id\_j, mim\_id\_j, hpo\_id\_j, common\_hpo\_id, hpo\_id\_i, mim\_id\_i, ps\_id\_i

FROM hpo\_pairs\_hetero WHERE mim\_id\_i > mim\_id\_j);

/\* Second, in table **tmp2\_ddsn**, add the IC values to the shared common\_hpo\_id of tmp1\_ddsn. \*/

CREATE TABLE **tmp2\_ddsn**(ps\_id\_i TEXT, mim\_id\_i INT, hpo\_id\_i TEXT, common\_hpo\_id TEXT, ic REAL, hpo\_id\_j TEXT, mim\_id\_j INT, ps\_id\_j TEXT);

INSERT INTO tmp2\_ddsn

SELECT l.ps\_id\_i, l.mim\_id\_i, l.hpo\_id\_i, l.common\_hpo\_id, r.ic, l.hpo\_id\_j, l.mim\_id\_j, l.ps\_id\_j

FROM tmp1\_ddsn l

JOIN hpo2ic r ON l.common\_hpo\_id = r.hpo\_id;

/\* Third, in table **d\_dp**, assemble the definitive bipartite graph as d\_dp. Note that table bipartite (in Module 3) is the bipartite graph in which each D node is paired with the DP node(s) that directly annotate that D in HPO. In contrast, d\_dp (In the present Module 6) is the bipartite graph in which each D node is paired with the DP (common\_hpo\_id) that the D shares with another D in the provisional DDSN (tmp2\_ddsn). \*/

CREATE TABLE **d\_dp**(mim\_id INT, hpo\_id TEXT, ic REAL);

INSERT INTO d\_dp

SELECT mim\_id\_i AS mim\_id, common\_hpo\_id AS hpo\_id, ic FROM tmp2\_ddsn

UNION

SELECT mim\_id\_j AS mim\_id, common\_hpo\_id AS hpo\_id, ic FROM tmp2\_ddsn;

DROP TABLE tmp; DROP TABLE tmp;

.once c:/SQLITE/LHPS/outputs/d\_dp.txt

SELECT \* FROM d\_dp;

/\* Fourth, in table **tmp3\_ddsn**, assemble the ‘preliminary’ version of the **DDSN-C**. The aim is achieved by assembling all the [Di>Dj] 2-tuples from tmp2\_ddsn, which are the nodes of the DDSN-C, while the [DP(i,j)] are the edge(s) linking the two D nodes from the [Di>DPi>DP(i,j)>DPj>Dj] 5-tuples of tmp2\_ddsn. It is preliminary, because each pair of D can be linked by more than one DP (one DP per edge), given that most of the pairs share more than one DP. \*/

CREATE TABLE **tmp3\_ddsn**(mim\_id\_i INT, common\_hpo\_id TEXT, ic REAL, mim\_id\_j INT);

INSERT INTO tmp3\_ddsn

SELECT DISTINCT \* FROM(

SELECT mim\_id\_i, common\_hpo\_id, ic, mim\_id\_j FROM tmp2\_ddsn);

**6.3 The definitive ‘non-redundant’ DDSN-C (one edge for the same node pair)**

/\* Fifth, in table **ddsn**, assemble the definitive (‘compact’) version of the DDSN-C, by grouping the [Di>DP(i,j)>Dj] 3-tuples of tmp3\_ddsn. Note that there cannot be the reverse pair (mim\_j and mim\_i), because the tmp1\_ddsn table was assembled with the criteria that mim\_ids are integers and that mim\_id\_i < mim\_id\_j. Also, as there are often more hpo terms annotating the same mim pair, calculate hpo term number (count\_hpo) and average of IC (ic\_mean) for all the shared hpo terms. \*/

CREATE TABLE **ddsn**(mim\_id\_i INT, mim\_id\_j INT, count\_hpo INT, ic\_mean REAL);

INSERT INTO ddsn

SELECT mim\_id\_i, mim\_id\_j, COUNT(DISTINCT common\_hpo\_id) AS count\_hpo, ROUND(AVG(ic),3) AS ic\_mean

FROM tmp3\_ddsn

GROUP BY mim\_id\_i, mim\_id\_j;

.once c:/SQLITE/LHPS/outputs/ddsn\_c.txt

SELECT mim\_id\_i, ic\_mean, mim\_id\_j FROM ddsn;