Spring 2013 CSCI-B669 Final Project

Counting 5 graphlets

Grace Law[*]
Indiana University
School of Informatics and Computing
Bloomington, IN
ylaw@indiana.edu

Ileane O'Leary[†]
Indiana University
School of Informatics and Computing
Bloomington, Indiana
ileane.oleary@hmc.edu

1. INTRODUCTION

Brute force counting of induced subgraphs (graphlets) is inefficient when the graphlets are larger than four nodes and the graph is large. This limits the application of graphlet counting to fields such as computational biology, where the data sets are huge and the graphlets have size over four. Developing efficient algorithms to count induced subgraphs (graphlets) in large graphs is necessary to foster new applications. Here we explore a novel algorithm, in which we store the locations of small graphlets, and then find larger graphlets by searching for two small graphlets that form the result.

2. LITERATURE REVIEW

Given a graph $G = (V, E)$, Marcus and Shavitt [1] presented algorithms to count all non-induced 4-graphlets of G rooted at each node $v \in V$. The term "non-induced" means a subgraph is counted even if additional edges exist between the nodes of the subgraph in G. Their algorithm has time complexity $O(d|E|+|E|^2)$, where d is the maximum nodal degree in graph G and $|E|$ is the number of edges in G. They then present a method to calculate the induced 4-graphlets count, given the corresponding non-induced 4-graphlet count. In this scheme, the only information that needs to be stored is the set of nodes adjacent to each one. The algorithm iterates over the edges and thus performs well on sparse graphs.

We would like to extend this idea to count graphlets of five or more nodes, with the aid of storing particular types of pre-computed smaller (3- and 4-) graphlet instances. For now, we will ignore the "roots" of the graphlets.

3. PROBLEM DEFINITION

Definition 3.1. *(Graph) A graph G is a pair of nodes and edges (V, E). Here we only consider undirected graphs with no labelling.*

Definition 3.2. *(Graphlet) A graphlet is a small (in terms of number of nodes) connected undirected unlabelled graph, up to graph isomorphism. A graphlet with n nodes is called an n-graphlet. Figure 1 shows all the non-isomorphic 5-graphlets. There are 21 of them.*

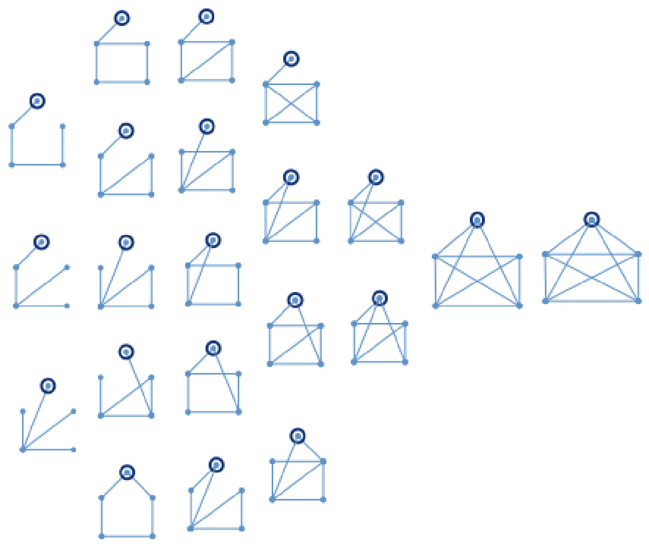


Figure 1: All non-isomorphic 5-graphlets. Ignore the small circles on particular node.

Definition 3.3. *(Induced and non-induced graphlet count) A subgraph (V_g, E_g) with $|V_g| = n$ of $G = (V, E)$ is said to be an induced graphlet of G if the inclusion map $f : V_g \to V$ satisfies the following condition:*

$$(u, v) \in E_g \text{ if and only if } (f(u), f(v)) \in E.$$

In case the above "if" direction is not satisfied, we call (V_g, E_g) a non-induced graphlet of G.

Our goal is to give an exact count of all induced 5-graphlets in a graph $G = (V, E)$.

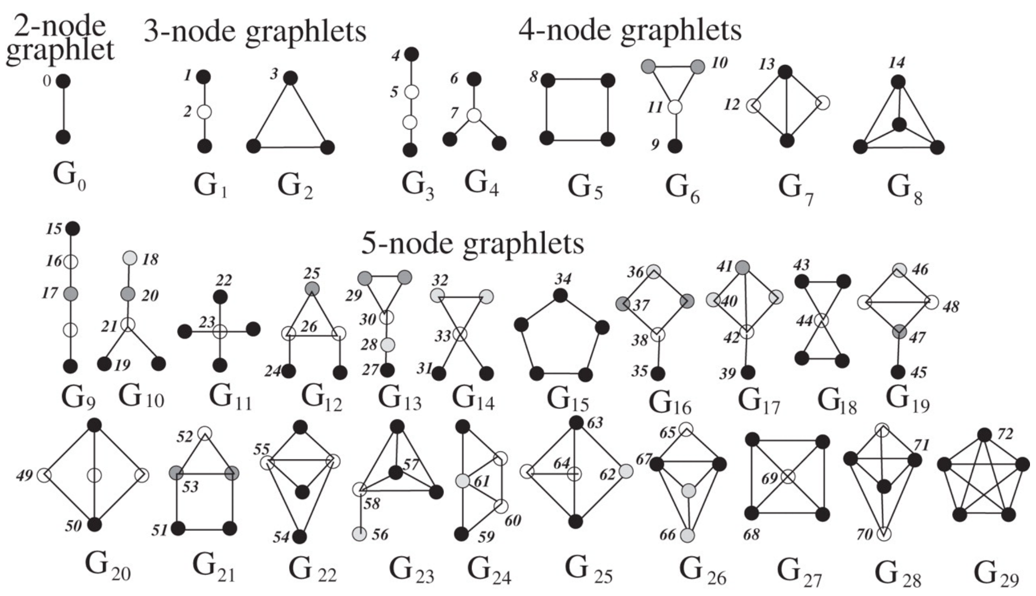


Figure 2: Naming convention of 2-, 3-, 4-, and 5-graphlets.

We are going to follow the naming convention of 5-graphlets as in Figure 2 (taken from [2]).

4. STRATEGY

Below we introduce some basic idea of our strategy.

4.1 What to store in the database?

Storing all instances of n-graphlets for $n < 5$ definitely helps in counting 5-graphlets. However, storing this much information takes too much space, especially if we later generalize our algorithm to counting graphlets with six or more nodes.

Instead of storing all possible n-graphlets, we store all paths P_n, claws S_n, cycles C_n and cliques K_n. This saves storage space. It can be verified through enumeration that all graphlets with fewer than 5 nodes can be created by joining graphlets from these sets.

Figure 3 shows the graphlets that are stored in order to generate $n = 5$. Note that 3-cliques and 3-cycles are the same. While 2-clique, 2-cycle, 2-claw and 2-path all refer to a single edge.

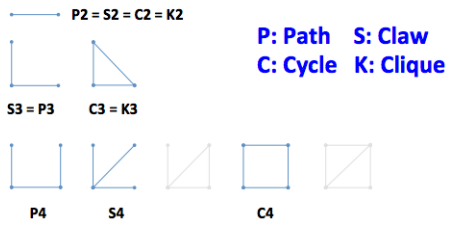


Figure 3: The path, claw, cycle and clique graphlets of 2, 3 or 4 nodes

The claw (S_4) instances mainly helps in counting the simple 5-graphlets which contain nodes with nodal degree 1.

4.2 Building 5-graphlets from smaller ones

We can use instances of

- P_2 to count all 3-graphlets;
- P_2, $P_3 = S_3$, $C_3 = K_3$ to count all 4-graphlets;
- P_2, P_3, C_3, P_4, S_4, C_4, K_4 to count all 5-graphlets;
- P_2, P_3, C_3, P_4, S_4, C_4, K_4, P_5, S_5, C_5, K_5 to count all 6-graphlets;

and so on.

Before we show how to join two graphlets, we need to define two terms: *positive conditions* and *negative conditions*. A positive or negative condition is a property of the graphlet. For any two nodes x and y in graphlet A, if x and y are adjacent, it is a positive condition. If x and y are not adjacent it is a negative condition. Since A is induced, an edge or lack thereof between x and y implies that the edge does or does not exist in the original graph.

For example, figure 4 shows the two possible 5-graphlets that could be generated by "joining" two K_4 graphlets. In this example, all the conditions are positive conditions because K_4 is complete.

Figure 5 shows that K_4 joining with C_4 would not produce any 5-graphlet, because the positive and negative conditions imposed by the two graphlets conflict with each other when joining. The dash lines shows the negative conditions: A C_4 graphlet instance should not contains the two dash red lines.

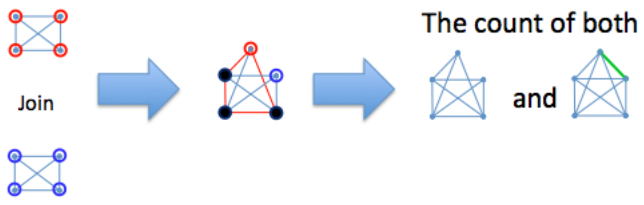


Figure 4: Joining two K4's

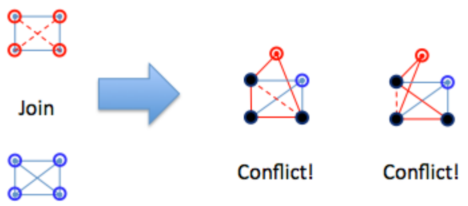


Figure 5: Joining a K4 with a C4

5. JOINING TWO GRAPHLETS

In this section we introduce an algorithm that given the inputs

1. an integer $N > 2$
2. two graphlets A and B (not necessarily different) from the set

$$\{P_n, S_n, C_n, K_n \,|\, 2 \leq n < N\}$$

returns all the possible N-graphlets in terms of their adjacency matrices and their names (as in figure 2) obtained by joining graphlets A and B. It also returns the corresponding SQL queries for joining the two graphlet instances tables for A and B.

The algorithm uses adjacency matrix of the graphlets to do calculations. Given the two input graphlet A and B we merge their adjacency matrices into one N by N matrix by assuming the two input graphlets have one or more shared nodes. The merged matrix will have some unfilled entries being the degrees of freedom. These unfilled entries represent edges between nodes in A and B which are not shared. Finally we output the graphlet names that corresponds to the resulting N by N matrix.

The algorithm works for $N = 3, 4, 5$ but can be easily generalized into $N \geq 6$ with additional information added into the hard coded hash functions.

5.1 Some assumptions in storing special graphlet instances

Suppose that in the given data graph, each node has a node id (nid) $\in \mathbb{N}$.

Assuming that we are given the tables of $\{P_n, S_n, C_n, K_n \,|\, 2 \leq n < N\}$ graphlet instances to begin with, we have some assumptions on how these graphlet instances are to be stored. Referring to figure 6, we assume that:

1. In storing a P2 instance, $nid(a) < nid(b)$.
2. In storing a P3 instance, $nid(b) < nid(c)$ and a is the only node that has nodal degree 2.
3. In storing a C3 instance, $nid(a) < nid(b) < nid(c)$.
4. In storing a P4 instance, a and d are the two nodes with nodal degree 1 and $nid(a) < nid(d)$. $b \in nbhd(a)$ and $c \in nbhd(d)$
5. In storing a S4 instance, $nid(b) < nid(c) < nid(d)$ and a is the only node that has nodal degree 3.
6. In storing a C4 instance, $nid(a) = min\{nid(k) \,|\, k = a, b, c, d\}$ and $nid(b) < nid(d)$ and c is the node opposite to a.
7. In storing a K4 instance, $nid(a) = min\{nid(k) \,|\, k = a, b, c, d\}$ and $b, d \in nbhd(a)$ and $nid(b) < nid(d)$.

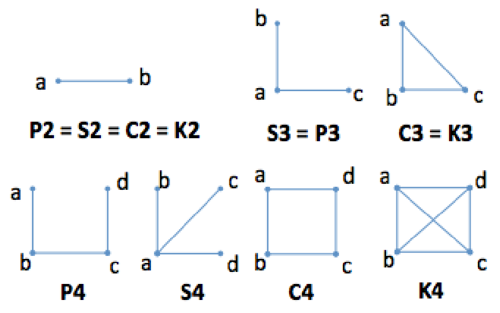


Figure 6: Assumptions when storing graphlet instances

5.2 An Example

We use an example to illustrate the algorithm. Suppose we input $A = S_4$ and $B = C_4$ and $N = 5$.

Let $n[graphlet]$ be the no. of nodes of the graphlet and $adj[graphlet]$ be the adjacency matrix of the graphlet.

Note that n[S_4] and n[C_4] are both 4. Since we are calculating all the result graphlets with 5 nodes, the number of nodes A and B must share is $4 + 4 - 5 = 3$. (That is, between A and B, there are 8 nodes, and to consolidate that to 5 nodes, 3 nodes of A must be the same as 3 nodes in B). These 3 nodes of A and B will be referred to as *join nodes*.

$adj[S4]$ is

$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \tag{1}$$

and $adj[C4]$ is

$$\begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} \tag{2}$$

We need to pick n[S_4]-3 = 1 node from graphlet S_4 to be an *outsider node*, that is, not a joined node. There are 4 possibilities to pick one node from 4, as shown in figure 7.

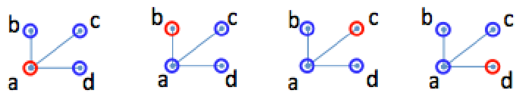


Figure 7: Picking an outsider in S4

For each such possible outsider, consider the minor of the matrix n[S_4] by removing the corresponding rows and columns.

e.g. If node a is the outsider, the minor is

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{3}$$

, indicating that the remaining 3 nodes b, c, d are all disconnected.

e.g. If either one of node b or c or d is the outsider, the minor is

$$\begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} \tag{4}$$

Now, let us look at C_4. Pick 3 nodes from C_4 to be joined with the 3 nodes in S_4. Consider the corresponding minor matrix of each such possible 3 nodes. In this case, no matter which 3 nodes we choose, the minor is going to look like:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \tag{5}$$

which indicates the existence of edges among the three blue nodes in figure 8. (Without loss of generality, suppose the three joining nodes are b, c, d.)

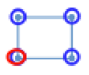
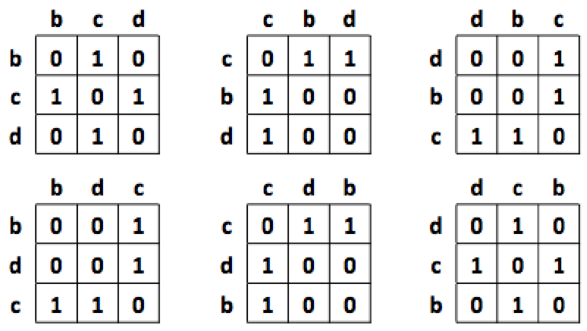


Figure 8: Picking three glueing nodes in C4

Notice that in minor (5), created from C_4, there are 2 edges. But, minor (3), from S_4 has 0 edges. Since these two minors do not have the same number of edges, we immediately know they are not equivalent. Thus, we can remove (3) from the list of S_4 minors to consider.

Now by permuting the three gluing nodes in C_4, (i.e. changing the way how they pair up with the gluing nodes in S_4) we obtain 6 possible matrices as shown in figure 9.



	b	c	d
b	0	1	0
c	1	0	1
d	0	1	0

	c	b	d
c	0	1	1
b	1	0	0
d	1	0	0

	d	b	c
d	0	0	1
b	0	0	1
c	1	1	0

	b	d	c
b	0	0	1
d	0	0	1
c	1	1	0

	c	d	b
c	0	1	1
d	1	0	0
b	1	0	0

	d	c	b
d	0	1	0
c	1	0	1
b	0	1	0

Figure 9: Permutations of gluing nodes in C4 result in different minors

Only the middle two permuted minors matches with the minor matrix (4). Figure 10 summaries the above steps.

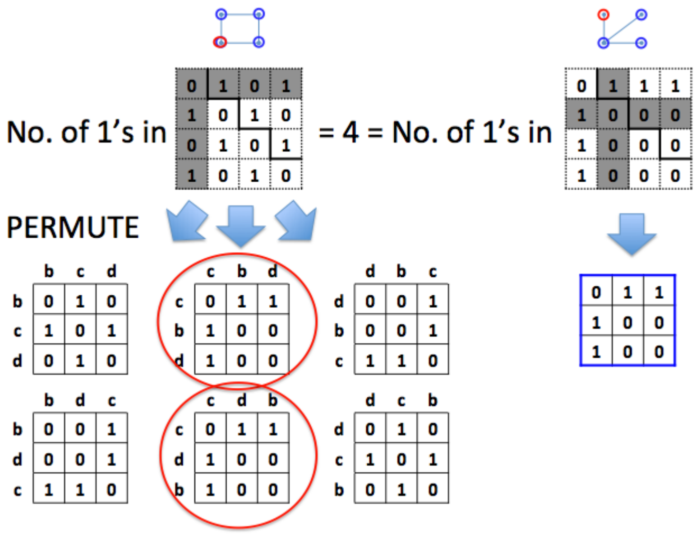


Figure 10: Finding matching minors of S4 and C4

Now we can merge the two adjacency matrices adj[S_4] and adj[C_4], following the order of permutation that we obtained above. For graphlet A, the order of the nodes is *bacd*, *cabd*, or *dabc*. For graphlet B, the order of the nodes is *cbda* or *cdba*. Figure 11 shows the merged 5 by 5 adjacency matrix created using order *bacd* for A, and *cbda* for B.

Figure 11 shows the corresponding merged graph. The red dashed edges are those edges that must not exist, because they are *negative conditions*.

Note that there are holes after merging the two matrices, represented by the green entries in figure 11. This corresponds to the green edge in figure 11. Note that a free position only occurs between a node from A and a node from B, where the two nodes are not shared.

Thus, there are two possibilities - the freedom positions may

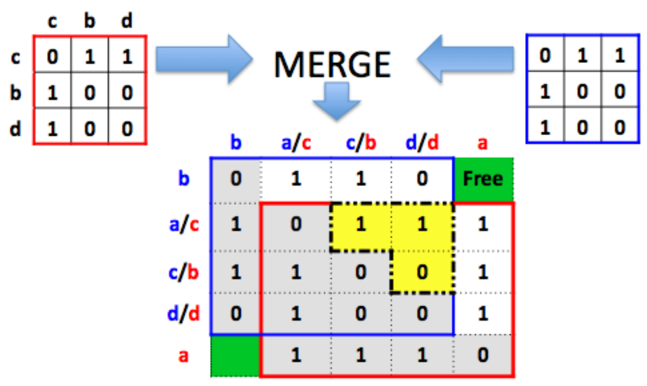


Figure 11: Merged adjacency matrix

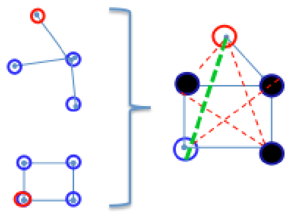


Figure 12: Merging graph, with 1 degree of freedom

be filled with 0's, representing the absence of the green edge, or 1's, representing the presence of the green edge. Therefore the two possible 5 graphlets after joining S_4 and C_4 are G_{16} and G_{20}, as shown in figure 13.

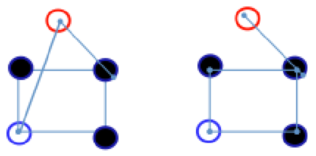


Figure 13: The two possible 5 graphlets (G_16, G_20) generated by joining C_4 and S_4

With all the other sets of possible gluing nodes from C_4, we arrive at the SQL query on the C_4, S_4 and P_2 tables in figure 14 (for obtaining the G_{16} graphlet) and figure 15 (for obtaining the G_{20} graphlet).

5.3 The algorithm

Figure 16 shows a very simplified pseudo code of the algorithm. For the detailed code, please refer to the appendix.

It turns out that the process of generating hash functions from adjacency matrices to graphlet names takes the longest time.

5.4 From non-induced to induced counts

Marcus and Shavitt [1] first present algorithms for counting non-induced graphlets, then use a formula to calculate the induced graphlet count. However it is not an easy job to do the over-counting calculations once we consider n-graphlets with $n > 5$.

Currently, the algorithm given in this project is capable of generating queries to find induced graphlet counts by checking for the presence of edges. However looking at all the possible adjacency matrix merges in figure 17, we can see that sometimes the degree of freedom can be large (up to 4). That corresponds to joining the P_2 table (which stores the edges) 4 times which is expansive. This is especially time consuming when there is a negative condition - that is, we are checking that edges between two nodes *do not* exist. In this case, we have to check every edge in the P_2 table, for all four possible-edges.

There are two suggestions to overcome this defect: either generate the non-induced graphlet queries and find a formula to handle the over-countings; or avoid using the joins that produce large degree of freedom.

We are able to find some of the induced graphlet counts using different joining results. For example we know that

1. K_4 join K_4 gives the count of both G_{28} and G_{29} graphlets;
2. K_4 join P_3 gives the count of G_{23}, G_{26}, and G_{28} graphlets;

Then we can take the intersection of these two join results to obtain the induced G_{28} graphlet count.

Notice that although joining other small graphlets may also result in a G_{28}, the above counts already covers all possible G_{28} instances. Since every instance of G_{28} graphlet should contains two different K_4 instances and those two K_4 instances should exist in the K_4 instance table.

5.5 Counting an induced graphlet multiple times

Another issue is counting the same induced graphlet more than once, since it is formed by two shapes in multiple ways. For example, consider G2, as constructed using P2 and P2. When we run the algorithm with $A = P_2$, $B = P_2$, and $N = 3$, we generate SQL code which describes every way the two $P2$ graphlets could join to G_2. But, G2 can be generated by P_2 and P_2 in 6 ways. First, the two $P2$ graphlets can be joined at any of the three corners (with the third edge as the freedom position), as shown in figure 17. Second, either of the edges can be selected as A, and the remaining one as B. Thus, if we run the SQL code generated, each G_2 graphlet will be returned 6 times, instead of only once.

There are several ways to handle this. One way is to determine which SQL queries are necessary to find all all graphlets exactly once. The advantage of this is that not only can we count the number of instances of a graphlet occurring in the graph, but we can also save the location of the graphlet without further processing (to remove the additional copies returned by the search). Determining which queries fulfil this property is an area for further work.

We handled this situation by calculating the number of copies we find during a particular search. The algorithm for this

```
SELECT  C4 .*,  S4 .*
FROM  C4 as A,  S4 as B, P2 as E
WHERE ( A. 0  = B. 2 AND  A. 2  = B. 1 AND  A. 3  = B. 3 AND ( 1 ,  2 )  NOT IN E )
OR ( A. 0  = B. 3 AND  A. 2  = B. 0 AND  A. 3  = B. 2 AND ( 1 ,  3 )  NOT IN E )
OR ( A. 0  = B. 0 AND  A. 2  = B. 1 AND  A. 3  = B. 3 AND ( 1 ,  0 )  NOT IN E )
OR ( A. 0  = B. 1 AND  A. 2  = B. 0 AND  A. 3  = B. 2 AND ( 1 ,  1 )  NOT IN E )
OR ( A. 0  = B. 2 AND  A. 1  = B. 1 AND  A. 3  = B. 3 AND ( 2 ,  2 )  NOT IN E )
OR ( A. 0  = B. 3 AND  A. 1  = B. 0 AND  A. 3  = B. 2 AND ( 2 ,  3 )  NOT IN E )
OR ( A. 0  = B. 0 AND  A. 1  = B. 1 AND  A. 3  = B. 3 AND ( 2 ,  0 )  NOT IN E )
OR ( A. 0  = B. 1 AND  A. 1  = B. 0 AND  A. 3  = B. 2 AND ( 2 ,  1 )  NOT IN E )
OR ( A. 0  = B. 2 AND  A. 1  = B. 1 AND  A. 2  = B. 3 AND ( 3 ,  2 )  NOT IN E )
OR ( A. 0  = B. 3 AND  A. 1  = B. 0 AND  A. 2  = B. 2 AND ( 3 ,  3 )  NOT IN E )
OR ( A. 0  = B. 0 AND  A. 1  = B. 1 AND  A. 2  = B. 3 AND ( 3 ,  0 )  NOT IN E )
OR ( A. 0  = B. 1 AND  A. 1  = B. 0 AND  A. 2  = B. 2 AND ( 3 ,  1 )  NOT IN E )
```

Figure 14: Resulting SQL queries on C_4, S_4 and P_2 tables to get the G_{16} graphlet

```
SELECT  C4 .*,  S4 .*
FROM  C4 as A,  S4 as B, P2 as E
WHERE ( A. 0  = B. 2 AND  A. 2  = B. 1 AND  A. 3  = B. 3 AND ( 1 ,  2 )   IN E )
OR ( A. 0  = B. 3 AND  A. 2  = B. 0 AND  A. 3  = B. 2 AND ( 1 ,  3 )   IN E )
OR ( A. 0  = B. 0 AND  A. 2  = B. 1 AND  A. 3  = B. 3 AND ( 1 ,  0 )   IN E )
OR ( A. 0  = B. 1 AND  A. 2  = B. 0 AND  A. 3  = B. 2 AND ( 1 ,  1 )   IN E )
OR ( A. 0  = B. 2 AND  A. 1  = B. 1 AND  A. 3  = B. 3 AND ( 2 ,  2 )   IN E )
OR ( A. 0  = B. 3 AND  A. 1  = B. 0 AND  A. 3  = B. 2 AND ( 2 ,  3 )   IN E )
OR ( A. 0  = B. 0 AND  A. 1  = B. 1 AND  A. 3  = B. 3 AND ( 2 ,  0 )   IN E )
OR ( A. 0  = B. 1 AND  A. 1  = B. 0 AND  A. 3  = B. 2 AND ( 2 ,  1 )   IN E )
OR ( A. 0  = B. 2 AND  A. 1  = B. 1 AND  A. 2  = B. 3 AND ( 3 ,  2 )   IN E )
OR ( A. 0  = B. 3 AND  A. 1  = B. 0 AND  A. 2  = B. 2 AND ( 3 ,  3 )   IN E )
OR ( A. 0  = B. 0 AND  A. 1  = B. 1 AND  A. 2  = B. 3 AND ( 3 ,  0 )   IN E )
OR ( A. 0  = B. 1 AND  A. 1  = B. 0 AND  A. 2  = B. 2 AND ( 3 ,  1 )   IN E )
```

Figure 15: Resulting SQL queries on C_4, S_4 and P_2 tables to get the G_{20} graphlet

```
def main(A, B, N)
    nglue <- calculate no. of nodes that need to glue
    Aoutsider <- calculate no. of outsider nodes from A
    A_outsider_candidates <- select_outsider(A, Aoutsider)
    for A_outsider_nodes in Aoutsider_candidates:
        rest_adjA = minor(adj[A], A_outsider_nodes)
        B_glue_candidates = select_B_glue(B, nglue ,rest_adjA)
        for B_glue_nodes in B_glue_candidates:
            merge_adj = merge_adj_matrix(A, A_outsider_nodes, B, B_glue_nodes, nglue, N)
            [freedom_positions, filled_adj_matrix, choice] = fill_in_freedom(merge_adj,N)
            SQL <- pruneSQL(A,B,remainder(n[A], A_outsider_nodes), B_glue_nodes)
            group the SQL's by corresponding graphlet names
    return set of resulting [graphlet names, set of corresponding SQLs]
```

Figure 16: Main algorithm

involves permuting all the rows and columns of the result matrix. For each permutation, check if the two joined matrices can be imposed in the result matrix permutation. So, check if the first matrix is in the upper left of the result matrix permutation, and the second matrix is in the lower left of the result matrix permutation. Count how many of these permutation have the joined matrix one and two in them, call this number *numCopy*.

Then, consider the permutations of matrix one and matrix two. This involves looking at how many permutations of matrix one are the same as the joined one, and likewise for matrix two. Call these numbers *factor1* and *factor2*. These are corrections factors. The final number of copies which we find during a particular search is $\frac{numCopy}{factor1 \cdot factor2}$.

This means for a single induced graphlet, we will find $\frac{numCopy}{factor1 \cdot factor2}$ copies when we search. To find the actual number of copies we take the number that we found in the search, and divide it by $\frac{numCopy}{factor1 \cdot factor2}$.

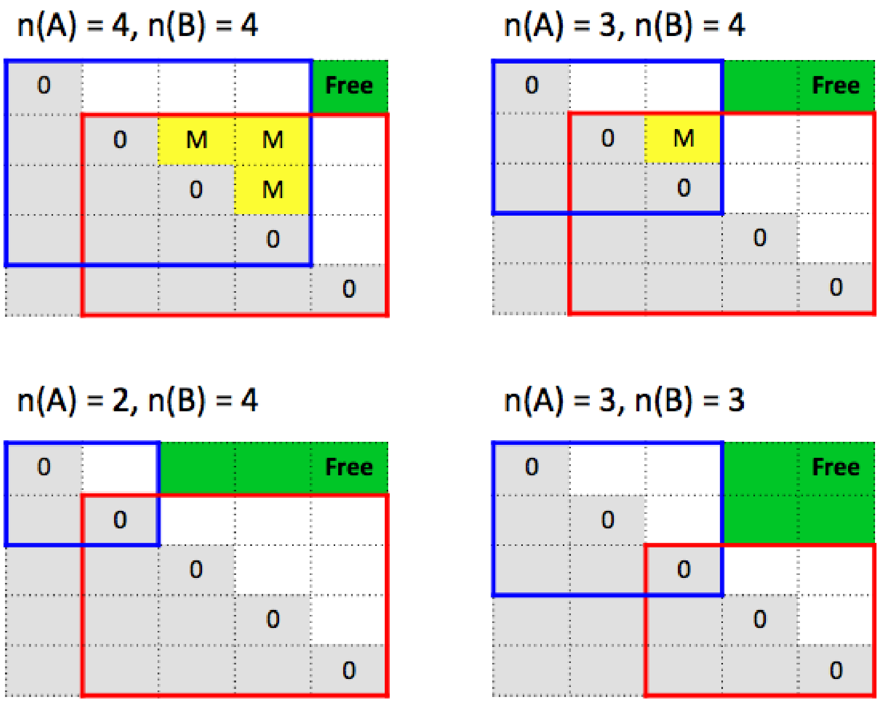


Figure 17: All possible adjacency matrix merges

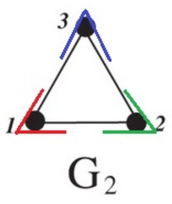


Figure 18: Each color represents one way to build G_2 from two P_2 graphlets, with the side opposite the joined P_2 graphlets as a freedom position.

6. FUTURE WORK

Eventually we want to use this search algorithm for graphlets of size larger than five. To do this, we'd want to save the result graphlets of the prior searches, so we could act recursively. For example, if we wanted to find a graphlet of size 7, we'd start with only a P_2 table, than use searches to find graphlets of size 3, 4, and 5. At that point, we should be able to use these graphlets to build the size 7 graphlets.

At this point, we still need to develop a good way to systematically output graphlet nodes of these result graphlets.

To improve the look-up times in our tables, we should consider building indices (e.g. the Triple-Tree index) on the database instances of $n < 4$ graphlets for efficient queries to facilitate 5-graphlets counting.

7. REFERENCES

[1] D. Marcus, Y. Shavitt, *RAGE - A rapid graphlet enumerator for large networks*, Comput. Netw. (2011), doi:10.1016/j.comnet.2011.08.019

[2] N. Przulj, *Biological Network Comparison Using Graphlet Degree Comparison*, Bioinformatics, v.23, n.2, p.e177-e183, 2007.

[3] Vladimir Vacic, Lilia Iakoucheva, Stefano Lonardi, and Predrag Radivojac, *Graphlet kernels for prediction of functional residues in protein structures*, Journal of Computational Biology. 17(1):55-72. (2010)

[4] Miroslaw Kowaluk, Andrzej Lingas, Eva-Marta Lundell. *Counting and detecting small subgraphs via equations and matrix multiplication.* In Dana Randall, editor, Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2011, San Francisco, California, USA, January 23-25, 2011. pages 1468-1476, SIAM, 2011.

[5] C. Augeri, B. Mullins, L. Baird, D. Bulutoglu, and R. Baldwin. *An algorithm for determining isomorphism using lexicographic sorting and the matrix inverse*, Congressus Numerantium, Utilitas Mathematica, 184:97■120, 2007.

[6] J. R. Ullmann, *An Algorithm for Subgraph Isomorphism*, Journal of the ACM (JACM), v.23 n.1, p.31-42, Jan. 1976.

8. APPENDIX

8.1 Some hash functions on graphlets

8.2 Hash: given an adjacency matrix output the corresponding graphlet name

8.3 Some helper functions

8.4 Some helper functions

```
import itertools
import timeit

start = timeit.default_timer()


changename = dict(P3='S3', C3='K3')

order = dict(P2=0, S3=1, K3=2, P4=3, S4=4, C4=5, K4=6)

n = dict(P2=2, S3=3, K3=3, P4=4, S4=4, C4=4, K4=4)
e = dict(P2=1, S3=2, K3=3, P4=3, S4=3, C4=4, K4=6)

#sym = dict(P2=[[0,1]], S3=[[0],[1,2]], K3=[[0,1,2]], P4=[[0,3],[1],[2]], S4=[[0],[1,2,3]], C4=[[0,2],[1,3]], K4=[[

store_nid_order = dict(P2=[[0,1]], S3=[[1,2]], K3=[[0,1,2]], P4=[[0,3]], S4=[[1,2,3]], C4=[[0,1,3],[0,2]], K4 = [[0

adj = dict(P2=matrix([[0,1],[1,0]]), S3=matrix([[0,1,1], [1,0,0], [1,0,0]]),
K3=matrix([[0,1,1], [1,0,1], [1,1,0]]), P4=matrix([[0,1,0,0], [1,0,1,0],
[0,1,0,1], [0,0,1,0]]), S4=matrix([[0,1,1,1], [1,0,0,0], [1,0,0,0], [1,0,0,0]]),
C4=matrix([[0,1,0,1], [1,0,1,0], [0,1,0,1], [1,0,1,0]]), K4=matrix([[0,1,1,1],
[1,0,1,1], [1,1,0,1], [1,1,1,0]]))
```

8.5 Some helper functions

8.6 The main function

8.7 Counting the number of times a graphlet is found

8.8 Output

```
def flatten(M, N):
    # turn a matrix M into row advance string
    L = []
    for i in range(N):
        for j in range(N):
            L.append(M[i][j])
    adjstr = "".join(str(x) for x in L)
    return adjstr


def generate_adj2graphlet_hash(N):
    # take all the isomorphic N graphlets, generate conjugacy class from it. Condition on elements.
    html('<!--notruncate-->')
    S_N = SymmetricGroup(N)
    equi_classes = []
    for g in graphs(N):
        if g.is_connected():
            #g.plot().show(figsize=[2,2])
            M = g.adjacency_matrix()

            equi_class = [M]
            for sigma in S_N:
                P = sigma.matrix()
                conju = P*M*P.inverse()

                inclass_already = False
                for element in equi_class:
                    if element == conju:
                        inclass_already = True
                        break

                if inclass_already == False:
                    equi_class.append(conju)

            equi_classes.append(equi_class)

    # hardcoded - follow Sage order.
    three_graphlet_names = [1, 2]
    four_graphlet_names = [4, 3, 6, 5, 7, 8]
    five_graphlet_names = [11, 10, 14, 9, 12, 16, 17, 20, 22, 15, 21, 24, 25, 26, 27, 28, 29, 13, 19, 23, 18]
    names = {3:three_graphlet_names, 4:four_graphlet_names, 5:five_graphlet_names}

    # building hash
    adj2graphlet = dict()
    count = 0
    for equi_class in equi_classes:
        thename = names[N][count]
        for M in equi_class:
            adjstr = flatten(M, N)
            adj2graphlet[adjstr] = thename
        count += 1

    return adj2graphlet
```

```
# n[graphlet] choose njoin
def select_outsider(graphlet, noutsider):
    possible = []
    L = list(itertools.combinations(range(n[graphlet]), noutsider))
    for i in L: possible.append(list(i))
    return possible


def minor(M, to_delete):
    # M is a square matrix (say k x k).
    # to_delete is a list of subset of range(k).
    # delete j-th row and j-th column for all j in to_delete
    v = remainder(M.nrows(), to_delete)
    B = M.matrix_from_rows_and_columns(v, v)
    return B


def select_B_join(B, njoin, M):   # return an ordered list of join nodes from B
    # B is graphlet B
    # njoin = no. of nodes to join
    # M = the joining part of adj matrix
    result = []

    Boutsider = n[B] - njoin
    outsider_candidates = select_outsider(B, Boutsider)

    for outsider in outsider_candidates:
        rest_adjB = minor(adj[B], outsider)

        v = remainder(n[B], outsider)

        if sum(sum(rest_adjB)) == sum(sum(M)):         # sum of all entries in the matrix
            real_permutes = Permutations(v).list()     # like permuting 0,2,3. This is the real index of node in B.
            vv = range(len(v))
            virtual_permutes = Permutations(vv).list() # like permuting 0,1,2. Need this for the adj matrix rest_adj

            for i in range(len(virtual_permutes)):     # len(virtual_permutes) = |v|!
                permute = virtual_permutes[i]
                permuted_adj = rest_adjB.matrix_from_rows_and_columns(permute, permute)
                if permuted_adj == M:
                    result.append(real_permutes[i])
    return result


def remainder(n, take_away_list):
    # returns the list [0,1,2,...,n-1] - take_away_list
    L = range(n)
    for out in take_away_list:
        L.remove(out)
    return L
```

```
"""
Input: A - matrix
       Aoutsider - list of nodes we are NOT joining from A
       B - matrix
       Bjoin - list of nodes that we are joinng from B
       N - dimension of merged result matrix
Output: the merged result matrix, with A in the upper right corner and B
        in the lower left. Free positions are marked with -1 entries.
"""
def merge_adj_matrix(A, Aoutsider, B, Bjoin, N):
    # reorder the list of nodes in A. outsider nodes followed by join nodes.
    Ajoin = remainder(n[A], Aoutsider)
    vA = Aoutsider + Ajoin
    reordered_adjA = adj[A].matrix_from_rows_and_columns(vA, vA)

    # reorder the list of nodes in B. join nodes followed by outsider nodes.
    Boutsider = remainder(n[B], Bjoin)
    vB = Bjoin+Boutsider
    reordered_adjB = adj[B].matrix_from_rows_and_columns(vB, vB)

    # Create the merged_adj, filled with -1's
    merged_adj = matrix(N, [-1]*N*N)

    # Place the reordered_adjA in the upper right corner of merged_adj
    merged_adj[0:n[A], 0:n[A]] = reordered_adjA

    # Place the reordered_adjB in the lower right corner of merged_adj
    merged_adj[(N - n[B]):N, (N - n[B]):N] = reordered_adjB

    return merged_adj



"""
Input: M - an adjacency matrix where freedom coordinates are marked as -1
       N - M is an NxN matrix
Output: a tuple, with
            a list of the location of the freedom positions (coordinates)
            a list of all possible adj matrices with freedom coordinates filled
"""
def fill_in_freedom(M,N):

    import copy

    #Find the location of all freedom positions
    freedom_positions = []
    deg_of_freedom = 0
    for col in range(N):
        # since the matrix is symmetrical, only look at the top half
        for row in range(col-1):
            if M[row,col] == -1:
                freedom_positions.append([row,col])
                deg_of_freedom += 1

    # generate a list of all possible ways to fill in the freedom coordinates
    zeroandone = [0,1]
    possibilities = list(itertools.product(zeroandone,repeat=deg_of_freedom))

    matrices = []
    # create matrices with all possible freedom coordinate combinations
    for choice in possibilities:
        temp = copy.deepcopy(M)
        for d, coord in zip(choice, freedom_positions):
            temp[coord[0],coord[1]] = d
            temp[coord[1],coord[0]] = d

        matrices.append([temp, choice])

    return [freedom_positions, matrices]
```

```
def pruneSQL(graphlet1, graphlet2, join1, join2):
    # join1 and join2 refers to the lists of joining nodes picked from graphlet1 & 2 resp.
    # e.g. S4 join S4. join1 = [2, 3], join2 = [3, 1]. i.e. 2<->3, 3<->1 when joining. graphlet1.col2 = graphlet2.co
    # Since in S4, nid(1)<nid(2)<nid(3), so the above join is not possible. Prune it.

    #store_nid_order = dict(P2=[[0,1]], S3=[[1,2]], K3=[[0,1,2]], P4=[[0,3]], S4=[[1,2,3]], C4=[[0,1,3],[0,2]], K4 =

    orders1 = store_nid_order[graphlet1]
    orders2 = store_nid_order[graphlet2]

    if len(join1) == 1:
        return 0

    possible_virtualpairs = list(itertools.combinations(range(len(join1)), 2))
    for virtualpairs in possible_virtualpairs:      #For every pair
        a1 = join1[virtualpairs[0]]
        a2 = join1[virtualpairs[1]]
        b1 = join2[virtualpairs[0]]
        b2 = join2[virtualpairs[1]]
        if (a2 - a1)*(b2 - b1) > 0:
            continue
        for order1 in orders1:
            if (a1 in order1) and (a2 in order1):
                for order2 in orders2:
                    if (b1 in order2) and (b2 in order2):
                        return 1
    return 0
```

```
def main(input1, input2, N):
    adj2graphlet = generate_adj2graphlet_hash(N)
    output_graphlets = []
    output_adjs = [] # adjacency matrices or out_graphlets
    output_SQLs = [] # ?

    # In case of equivalent names, change them to preferred version...
    if input1 == 'P3' or input1 == 'C3':
        input1 = changename[input1]
    if input2 == 'P3' or input2 == 'C3':
        input2 = changename[input2]

    # order the matrices as desired and asign them to A and B...
    if order[input1] <= order[input2]:
        A = input1
        B = input2
    else:
        A = input2
        B = input1

    njoin = n[A] + n[B] - N  # no. of nodes to join
    Aoutsider = n[A] - njoin  # no. of outsider/nonjoin nodes from A
    Aoutsider_candidates = find_outsider_sets(A, Aoutsider)

    for Aoutsider_nodes in Aoutsider_candidates: # for each outsider candidate
        rest_adjA = minor(adj[A], Aoutsider_nodes) # create minor matrix

        # Select joining candidates from B
        B_join_candidates = find_join_sets(B, njoin ,rest_adjA)
        for B_join_nodes in B_join_candidates:
            merge_adj = merge_adj_matrix(A, Aoutsider_nodes, B, B_join_nodes, N)
            # Only used in output.
            vA = Aoutsider_nodes + remainder(n[A], Aoutsider_nodes)
            # Only used in output.
            vB = B_join_nodes + remainder(n[B], B_join_nodes)
            # filled_list = [filled_adj_matrix, choice]
            [freedom_positions, filled_list] = fill_in_freedom(merge_adj,N)

            for tuple in filled_list:
                filled_adj_matrix = tuple[0]
                choice = tuple[1]
                # group result by graphlet names
                output_name = adj2graphlet[flatten(filled_adj_matrix)]
                if pruneSQL(A,B,remainder(n[A], Aoutsider_nodes), B_join_nodes)\
                                                                          == 1:
                    continue
                # output_graphlets is sorted, in ascending order
                if output_name in output_graphlets:
                    insert_place = output_graphlets.index(output_name)
                    output_adjs[insert_place].append(filled_adj_matrix)

                    # Here's where the output SQL's are made
                    sqlCode = sqlStatementInfo(remainder(n[A], Aoutsider_nodes),\
                                           B_join_nodes, freedom_positions,\
                                           choice, vA, vB)
                    output_SQLs[insert_place].append(sqlCode)
                else:
                    if len(output_graphlets) == 0: insert_place = 0
                    elif output_graphlets[-1] < \
                         output_name: insert_place = len(output_graphlets)
                    else:
                        for name in output_graphlets:
                            if name > output_name:
                                insert_place = output_graphlets.index(name)
                                break
                    output_graphlets.insert(insert_place, output_name)
                    output_adjs.insert(insert_place, [filled_adj_matrix])
                    # ouput SQL's are made here too
                    sqlCode = sqlStatementInfo(remainder(n[A], \
                                                     Aoutsider_nodes), \
                                           B_join_nodes, freedom_positions,\
                                           choice, vA, vB)
                    output_SQLs.insert(insert_place, [sqlCode])

    return [output_graphlets, output_adjs, output_SQLs, A, B]
```

```
"""
Input: adj1 - a graphlet
       adj2 - a graphlet
       graphletList - a list of graphlets,
           generated by taking the result graphlet of adj1 and adj2, and
           permuting the rows and columns
Output: the number of times the result graphlet will be found if you
        search for it using adj1 and adj2
"""
def numCopies(adj1, adj2, graphletList):

    # Calculate overlap dimensions
    oDim = adj1.nrows()+adj2.nrows()-graphletList[0].nrows()

    # Count the number of copies
    numCopy = 0
    for adj in graphletList:

        # check if the upper left corner is equal to adj1
        match = compareMatrixPortion(adj1, adj, 0,0)

        # check if the lower left corner is equal to adj2
        if match:
            sRow = adj.nrows() - adj2.nrows()
            sCol = adj.ncols() - adj2.ncols()
            match = compareMatrixPortion(adj2, adj, sRow, sCol)

        # increment if both matrices are there.
        if match:
            numCopy += 1

    # Calculate how many equivalent rotations of adj1
    factor1 = 0
    for p in permutations(range(0, oDim)):
        v = p + range(oDim, adj1.nrows())
        A = adj1.matrix_from_rows_and_columns(v,v)

        if adj1 == A:
            factor1 += 1

    # Calculate how many equivalent rotations of adj2
    factor2 = 0
    for p in permutations(range(oDim, adj2.nrows())):
        v = range(0, oDim)+p
        A = adj2.matrix_from_rows_and_columns(v,v)

        if adj2 == A:
            factor2 += 1

    # Removing duplicates, using the factors calculated:
    if factor1 == 0 or factor2 == 0:
        print "Error in function numCopies. factor1 or factor2 equals 0."
    else:
        numCopy = numCopy/(factor1*factor2)

    return numCopy
```

```
def saveSQLToFile(result_graphlets, result_adjs , result_SQLs, A, B):
    print "\n \n \n"
    fileName = "sqlCode" + A + "+" +B + ".txt"
    print "Saving SQL code to file. Filename: ", fileName
    o = open(fileName, 'w')
    o.write("/* Joining " + A + " and " + B + ". */ \n")

    # for each graphlet
    for i in range(len(result_graphlets)):
        o.write("\n")
        o.write("/*The graphlet is G"+str(result_graphlets[i])+"*/ \n")

        # look up the resulting adjacencies
        for j in range(len(result_adjs[i])):
            if j == 0:
                o.write("\n")
            else:
                o.write("UNION \n")

            # Look up the sql code for the graphlet and adjacency.
            sqlCode = result_SQLs[i][j]

            # Display all nodes in final thing exactly once!
            # First print all non-joining A nodes
            nonJoinA = remainder(n[A], sqlCode.Ajoin)
            o.write("SELECT ")
            o.write("A.n" + str(nonJoinA[0]))
            for k in nonJoinA[1:]:
                o.write(" , A.n" + str(k))
            # Then print all B nodes
            for k in sqlCode.Bnodes:
                o.write(" , B.n" + str(k))

            # Handle FROM clause
            o.write("\n") # new line after SELECT...
            o.write("FROM " + A +" as A ," \
                    + B + " as B \n")
            # Handle WHERE clause
            o.write("WHERE (")
            o.write("A.n" + str(sqlCode.Ajoin[0]) + \
                    " = B.n" + str(sqlCode.Bjoin[0])+ "\n")
            for nodeA, nodeB in zip(sqlCode.Ajoin[1:], sqlCode.Bjoin[1:]):
                o.write("AND A.n" + str(nodeA) + " = B.n" + str(nodeB)+ "\n")
            for nodeA in nonJoinA:
                for nodeB in sqlCode.Bnodes:
                    o.write("AND A.n"+str(nodeA) + " <> B.n" + str(nodeB)+"\n")

            # Handle freedom positions
            for node, choice in zip(sqlCode.freePos, sqlCode.choice):
                # Get an edge, indexA is the index of the node in
                # the list of total nodes in A, likewise with B
                indexA = node[0]
                indexB = n[B] - 1 - (N-1-node[1])


                if choice == 0:
                    # code for edges which are not allowed
                    o.write("AND NOT EXISTS \n")
                else:
                    # code for edges which must exist
                    o.write(" AND EXISTS \n")

                o.write("(SELECT * \n")
                o.write("FROM P2 as D \n")
                o.write("WHERE (A.n"+ str(sqlCode.Anodes[indexA]) + "= D.n0 ")
                o.write("AND B.n"+ str(sqlCode.Bnodes[indexB])+"= D.n1 ) \n")
                o.write("OR (A.n"+ str(sqlCode.Anodes[indexA]) + "= D.n1 ")
                o.write("AND B.n"+ str(sqlCode.Bnodes[indexB])+"= D.n0 ) ) \n")


            o.write(")") # closes parenthis started at "WHERE(...)"
    o.close
```