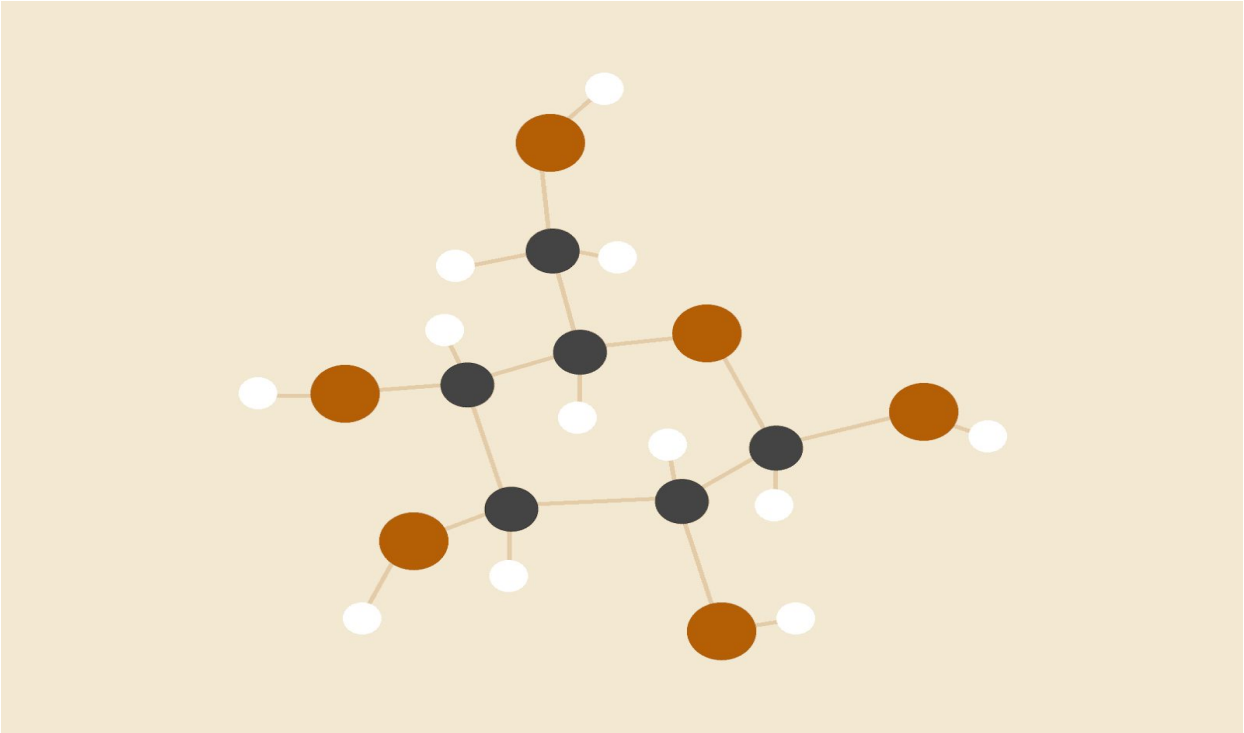
Provenance graph Summarization



Jigar Dalwadi

Under the guidance of

Dr. David Koop

01684802

CIS 600

Master's Project

1 INTRODUCTION

The idea of the project is make a dynamic provenance graph and make groups on various logic to summarize the graph . We are using dagre js to make graph. Provenance graph can be too large and for that reason it can be very hard to understand. Our goal is to find a solution which helps us to make groups of the nodes based on some similarities or based on some grouping algorithms. Making groups in the graph helps us to read and understand the graph/provenance data easily.

1.1 DATASET and LANGUAGE

The provenance data can be in various formats. We are getting the Prov data from Prov Store. We can get the prov data in different formats. The data can be in Prov-N, Json, XML and Turtle format. We are using Prov JSON format . The provenance data contains entities, activities and agents.

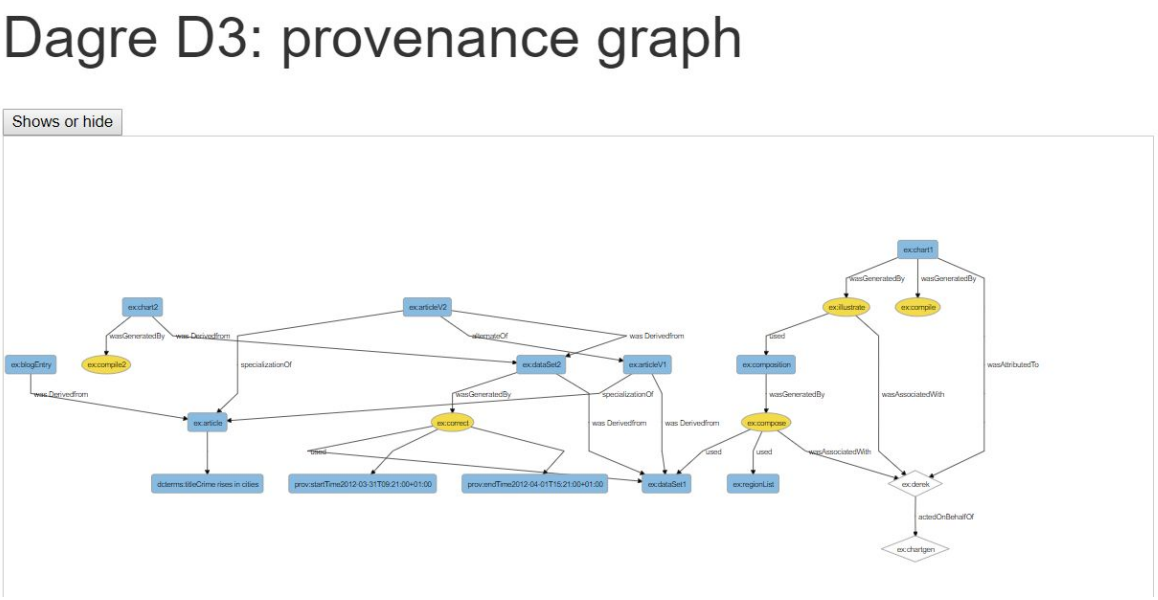
An **entity** is a physical , digital , conceptual, or other kind of thing with some fixed aspects; entities may be real or imaginary. An **activity** is something that occurs over a period of time and acts upon or with entities; it may include consuming, processing, transforming, modifying, relocating, using, or generating entities. An **agent** is something that bears some form of responsibility for an activity taking place, for the existence of an entity, or for another agent's activity.Entities and Activities can be connected with each other using “Generated by” and “Used”. Entity can be regenerated or transformed using “Derived by”. Activity and Agent can be connected using “wasAssociatedwith”. All this connections will be mentioned as the edge label in the graph.

The best way to draw and visualize the graph will be using D3 js. We are here using Dagre D3 to visualize the graph. Dagre has some inbuilt function like “g.setNode” , ”g.setEdge” and many more. We’ll use this inbuilt functions to create the graph. For making groups or summarized graph also we are going to use the different functions of Dagre d3 library.

2 TYPES OF PARTITIONING TECHNIQUES

We creating the graph using json file. Here i have created a javascript file which tries to create entities, activities, agents and different relations between them. After creating the graph I have added rendering and zoom in zoom out functions. Here I have implemented different strategies to summarize the graph. It's based on cardinality, parent-child relation, Common relation, Greedy approach and Refinement based on greedy approach.

Implementation of graph. To draw a graph i have used several functions which reads data from json file and converts the data into a graph. The provegraph.js file includes all the functions required to build the graph. If any kind of change we need in graph it can be done in provegraph js. There are functions called "createentity","activity","agent" and many more. Basically for every kind of node and edge there is a function which reads key: value pair from Json file. After including this javascript in your html page we'll be able to visualize the graph.



2.1 Cardinality based grouping

The basic idea behind this is to find the maximum outgoing edges from each node.

Then finding the maximum out of it. Grouping all those nodes which are having outgoing edges as the maximum are set into a one group. We can replace all those nodes and make a single node instead which will help to minimize the graph and also it makes easier to understand the large graph.

For example in in a graph we'll find out edge count for each node. Eg: node1: 2, node2: 2, node3 : 1, node4: 3, So in this example this countmaxedges function will return 2. So the nodes 1 and 2 will be placed in a same group.

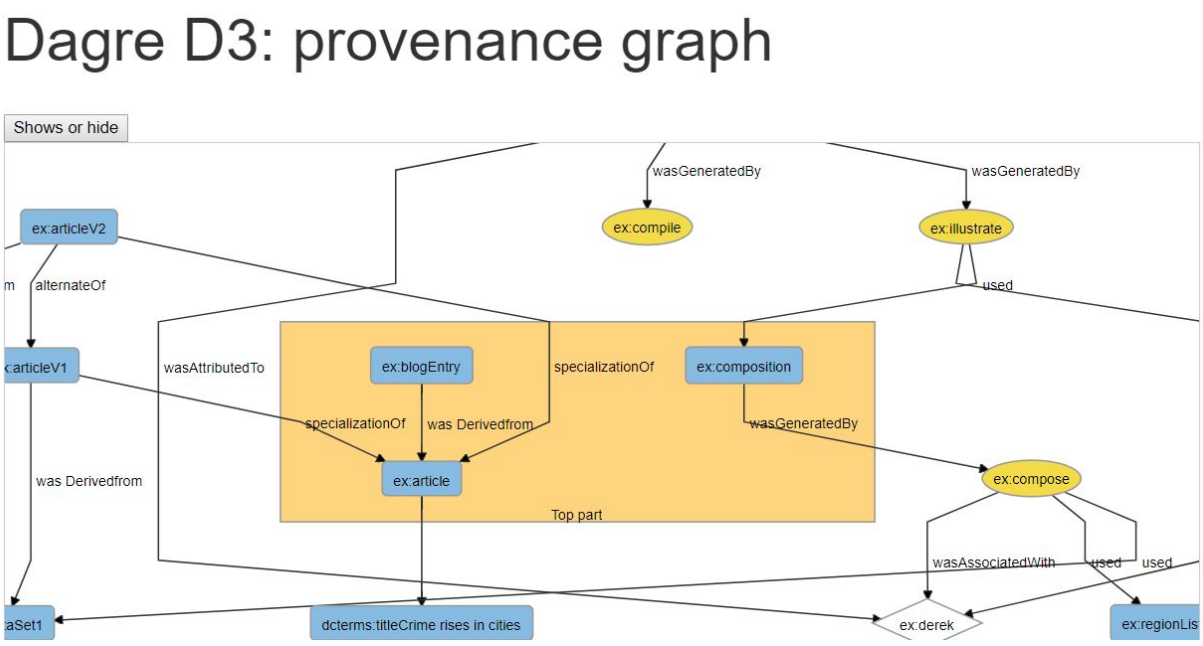
```
var MaximunOccurence = countmaxedfes();
var x1;
function countmaxedfes(){
var countarray = [];
    for(var entity in entityKeys){
    x1 = g.outEdges(entityKeys[entity]).length;
    countarray[entity] = x1;
    }
    var v = mode(countarray);
    function mode(array){
        if(array.length == 0)
            return null;
            var modeMap = {};
            var maxEl = array[0], maxCount = 1;
            for(var i = 0; i < array.length; i++){
                var el = array[i];
                if(modeMap[el] == null)
                    modeMap[el] = 1;
                else
                    modeMap[el]++;
                    if(modeMap[el] > maxCount)
                    {
                        maxEl = el;
                        maxCount = modeMap[el];
                    }
            }
        return maxEl;
    }
    return v;
}
```

2.1.0 Implementation for cardinality based grouping

The first step to implement it is to create a function which calculates maximum number of edges. In the algorithm it's calculated as a function countmaxedges(). This function returns a number. This number then used to compare with the outgoing edge number for all the nodes. Whichever nodes outgoing edge number matches to the result of countmaxedges will be placed into a group. The transformation from the original graph to summarized graph can be done using a button(only used in this algorithm). When we click on the button which makes the group. In a big graph this algorithm summarizes the graph very well.

Countmaxedges function will look the above image.

2.1.1 Output



2.1.2 Statistics

Graph number	Number of nodes	Number of edges	Countmaxedges	Nodes after grouping
article-prov 1.json	20	24	1	17
article-prov 2.json	78	219	2	27
article-prov 3.json	62	162	3	38

2.2 Parent-Child Relationship based grouping

The simplest way to group some nodes is based on their parents and child. If some nodes have same parents and those parent nodes also have no dependencies then we can make a group of those nodes. The other way is to check for the children. If two or more nodes have same parents and no dependencies then we can make a group of those nodes. For this the must thing we need to keep in mind is the dependencies. So most probably the nodes which are getting grouped are those which has no parents or those which has no children. By doing this we can group those node which looks like a loose end/start.

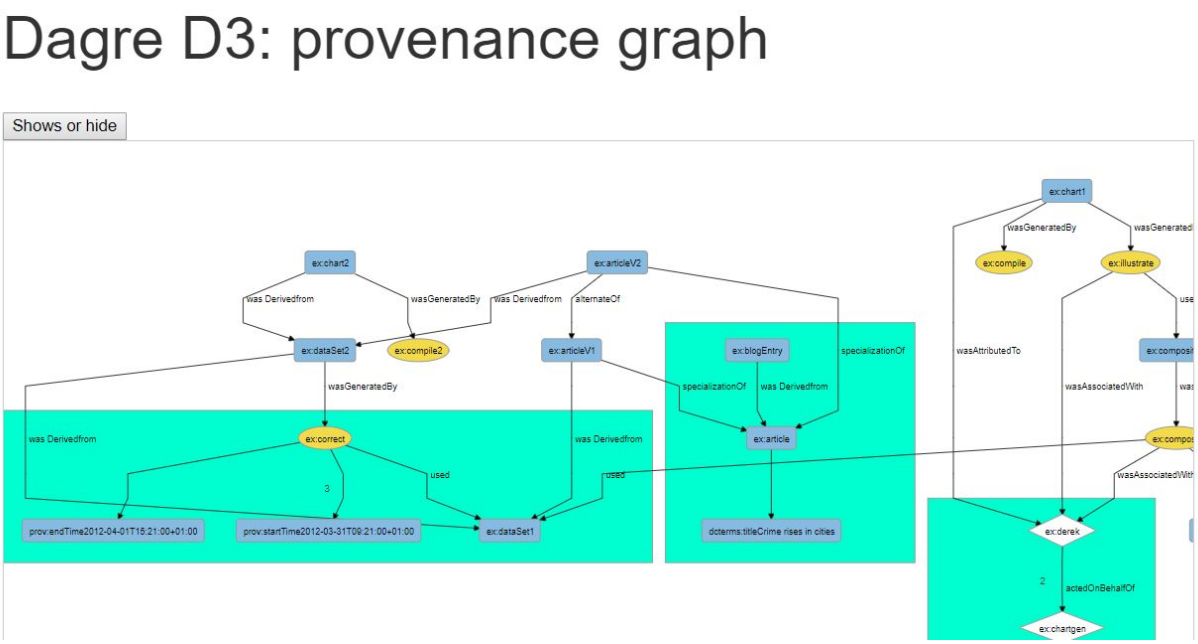
The main benefit from this is that we get a more clear graph as all the individual solo kind of nodes will be in the groups with their parent/child node.

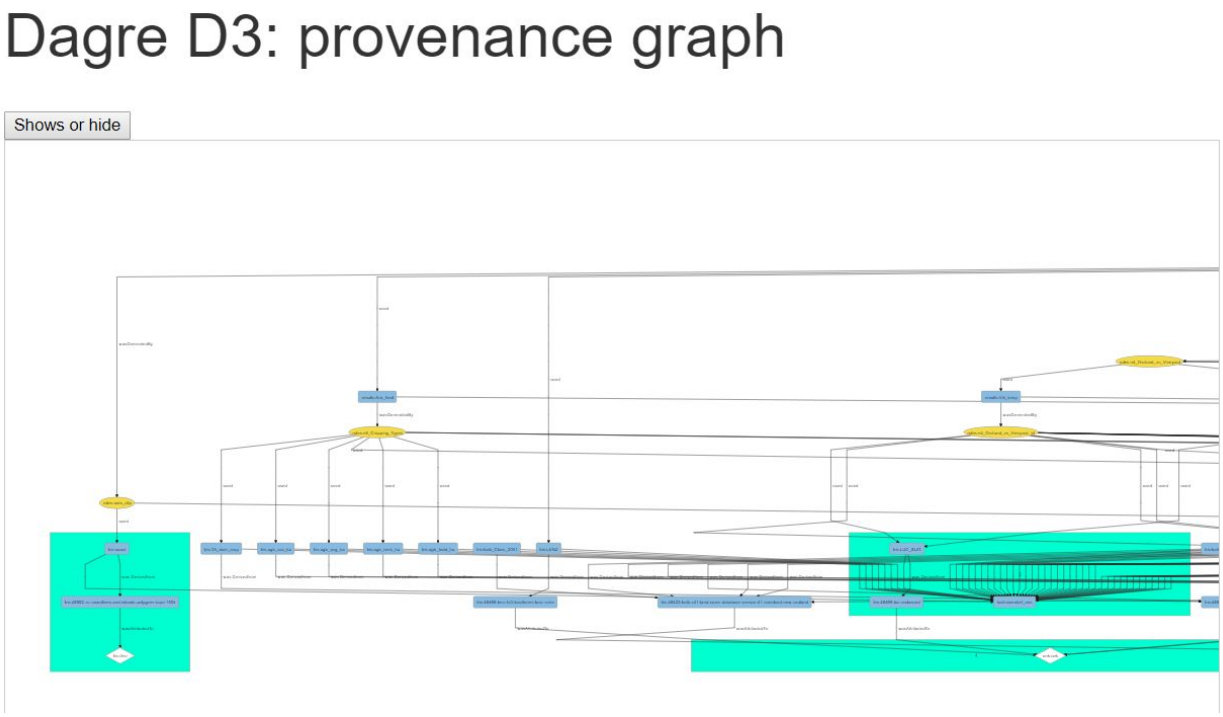
2.2.0 Implementation of Parent-Child Relationship based grouping

To implement this we need to find the parent of all the nodes and as well we need to find the children of all nodes. After we get all these values we are setting up an onclick event for all nodes. When we click on a particular node it checks for the parent of that node and also the children of that node. Whichever node has same parent or same children will be placed into a same group. One more thing we can find is the nodes with no outgoing edges and no incoming edges. We can use “g.predecessors” and “g.successors” to find those nodes. Then we can Compare those parents and children to make groups.

2.2.1 Output and statistics

The number of nodes in a group can vary according to the click on the nodes. It's decided after the click on the nodes. Even the edge cuts will depend on the number of nodes we click on.





2.3 Relationship based grouping

The idea of implementing this algorithm came from the paper on Efficient Aggregation for Graph Summarization. In the paper the algorithm SNAP and K-SNAP are used for finding the relations between nodes and making groups based on it. This is also called as similarity based grouping. In this grouping we are finding the relations of each node. After finding the relations of each node we have to compare them with each other. If we find similarities between two or more nodes then we can make a group of that nodes. In some graphs we see that so many nodes have same kind of incoming and outgoing edges. So by making a group of this nodes will make a graph small and more understandable.

2.3.0 Implementation of Relationship based grouping

The first thing we need to find is the outgoing edges for all the nodes. After finding number of outgoing edges we need to compare the edgelables with the other nodes outgoing edges labels. If the number of edges and also the labels are same then we'll make a group of those nodes. By doing this we are making groups of similar kind of nodes. They can have same parents or different parents and same for their childeren nodes.

To start with first we need are creating an array of an object which contains based on the g.predecessors and g.successors , which will have node name and the edgelables of the edges going out of each nodes.

```
function relations(){
for(var x in entityKeys){
    if(g.successors(entityKeys[x]) != undefined){ // if there is no succesor
        var edgelab =   [];
        for(var y in g.successors(entityKeys[x])){//find succesor and edgelable
        edgelab.push(g.edge(entityKeys[x],g.successors(entityKeys[x])[y]));
        }
        var obj1 =  {
            entity: entityKeys[x],
            vals : edgelab
        };
        lable1.push(obj1)
    }
}
for(var x in activityKeys){
    if(g.successors(activityKeys[x]) != undefined){ // if there is no succesor
        var edgelab =   [];
        for(var y in g.successors(activityKeys[x])){//find succesor and edgelable
        edgelab.push(g.edge(activityKeys[x],g.successors(activityKeys[x])[y]));
        }
        var obj2 =  {
            entity: activityKeys[x],
            vals : edgelab
        };
        lable2.push(obj2)
        console.log(lable2)
    }
}
```

We are going to do this for activity and entities nodes one by one. After this we'll get to arrays of object lable1 and lable2. We'll loop through all the objects in lable 1 and lable 2 to find the match between them. Who ever has same length and edgelabel will be stored in an array named storeactivity and storeentitie.

```
for(x=0; x < lable1.length;x++){
    var store1 = {
        entity1 : lable1[x].entity,
        entity2: []
    }
    for(y=x+1; y< lable1.length;y++){
        if(lable1[x].vals.length == lable1[y].vals.length){
            for(var z in lable1[x].vals){
                // if they are same
                if(lable1[x].vals[z].label && lable1[y].vals[z].label &&
                lable1[x].vals[z].label == lable1[y].vals[z].label){
                    //console.log("z  " +z)
                    //console.log("lab " +lable1[x].vals.length)
                    if(z == lable1[x].vals.length-1){
                        store1.entity2.push(lable1[y].entity);
                        lable1.splice(y, 1);
                        if(storeentity.indexOf(store1)<0){
                            lable1.splice(y, 1);
                            storeentity.push(store1);
                            //g.setNode(x,{label: x});
                        }
                    }
                }

            }
        }
    }
}
```

This above code will help us to make groups for entities. The same we have to do for activities also. After that we'll ahave two array named storeentity and storeactivity. Both will have nodes wich has same kind of relations.

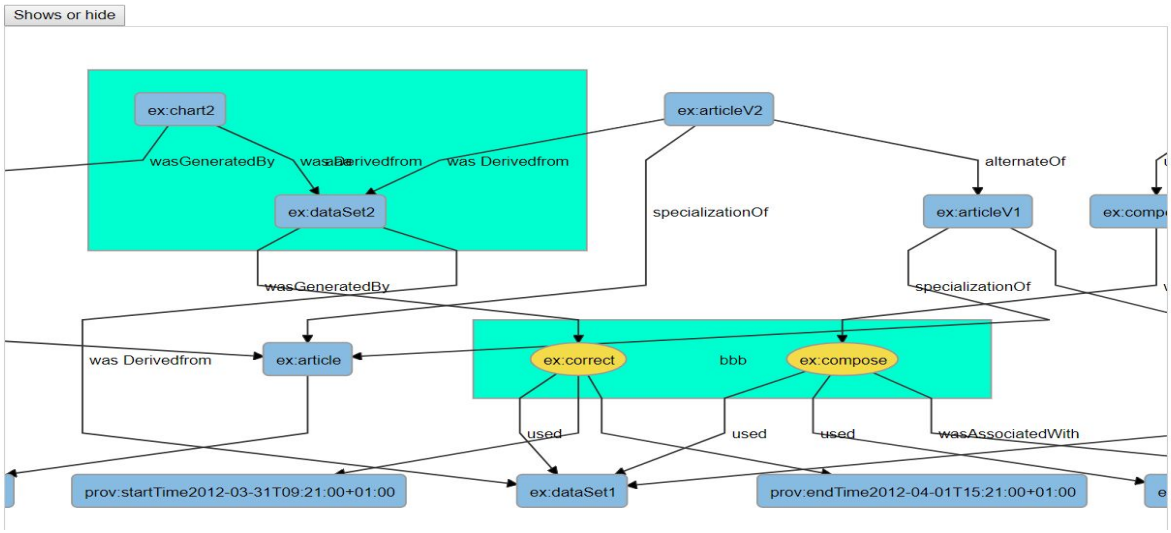
After we get these arrays we'll create groups and assign the nodes their parent nodes.

```
for(var x in storeentity){
    if(g.parent(storeentity[x].entity1) == undefined ){
        g.setNode(x,{label: "aaa"});
        g.setParent(storeentity[x].entity1,x);
        //console.log("adsad");
        for(var y in storeentity[x].entity2){
            g.setParent(storeentity[x].entity2[y],x);

        }

    }
}
for(var x in storeactivity){
    if(g.parent(storeactivity[x].entity1) == undefined ){
        var y = x+500;
        g.setNode(y,{label: "bbb"});
        g.setParent(storeactivity[x].entity1,y);
        for(var z in storeactivity[x].entity2){
            g.setParent(storeactivity[x].entity2[z],y);
        }
    }
}
init(g);
```

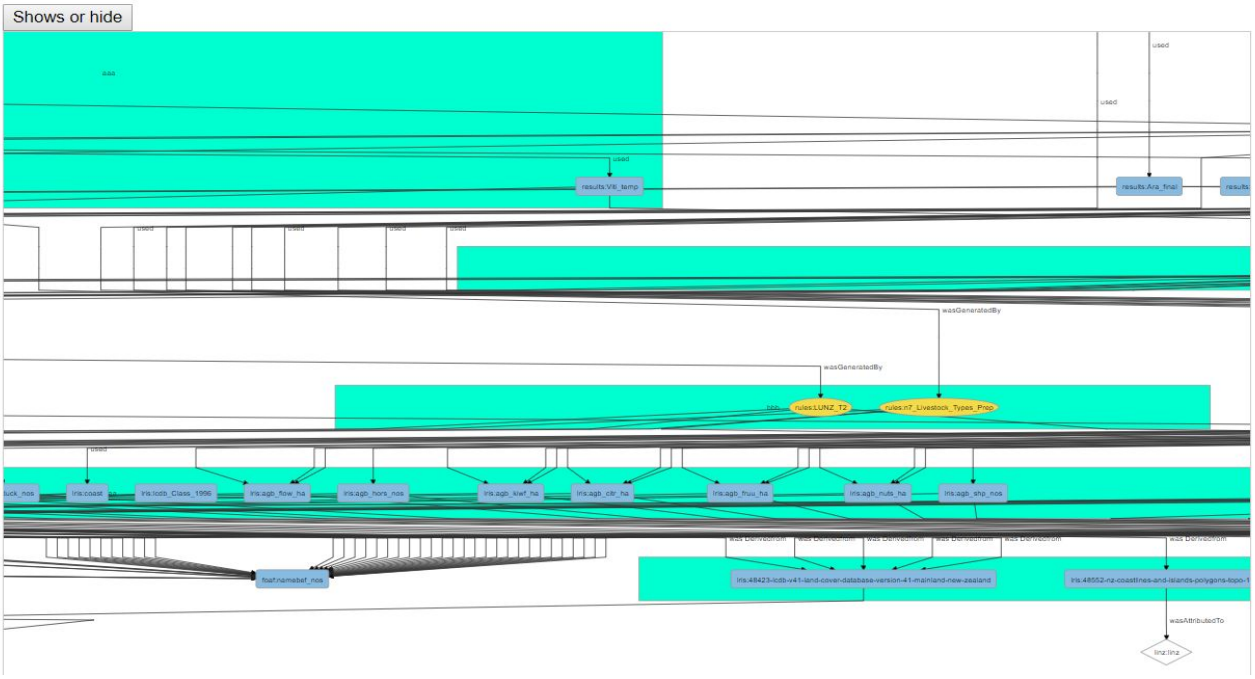
2.3.1 Output

We get multiple groups with minimum of 2 nodes in a groups upto as multiple nodes.

Dagre D3: provenance graph



Dagre D3: provenance graph



2.3.2 Statistics

Graph number	Number of nodes	Number of edges	Number of nodes got into groups
article-prov 1.json	20	24	4
article-prov 2.json	78	219	58
article-prov 3.json	62	162	44

2.4 Metis Graph Partitioning

Metis is a serial graph partitioning algorithm. Metis gives us many options to partition the graph. Here the input will be in a separate format. First of all we need to convert out data into metis input format. So when we load the metis.html it will create a metis input graph file. The format will have node numbers and its connections. For example node 1 is connected to node 2 and 3, so the input file will contain data as {1->2,3}. Just like this for every node its connections will be written in the file. After getting that file we can input that into metis algorithm which returns the node name with group number. After that we'll load that partitioned file into our metis.html file and reload it to get the partitioned graph.

2.4.0 Implementation of Metis Graph Partitioning with outputs and statistics

The first thing is to install metis. The instruction for the installing is given in the downloading package(6). After downloading when you'll run the gpmetis in visual studio it will create a gpmetis.exe file.
The steps to run the gpmetis:

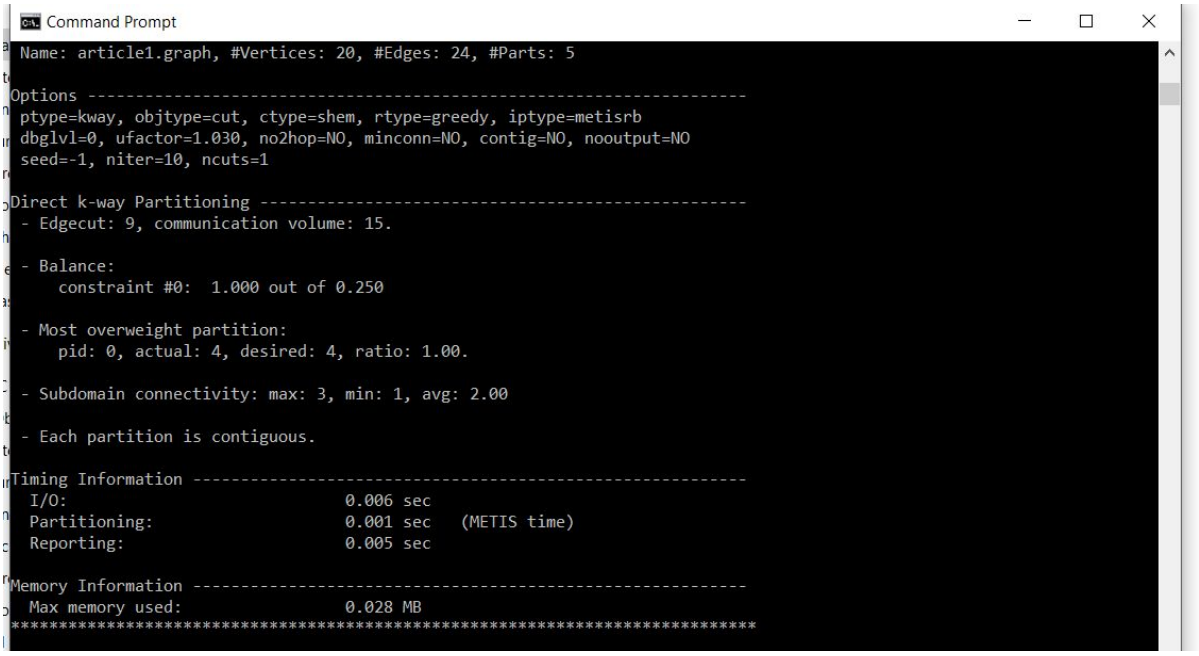
1. gpmetis -help // this checks if the gpmetis work properly.
2. gpmetis [OPTIONS] GRAPGFILE.GRAPH numberofpartitions

The list of options OPTION will be found when we'll run -help. We need to make a graphfile for the input in metis. The input file need to be in the metis format.

The .graph file will contain number of nodes and number of edges in the first line. The next line will be considered as the node number and it will contain the connecting nodes for each node.

```
20 24
2
1 8 9 20
11
6 13
16 14 12
4 9 10 15
12 11
9 10 2
6 8 2
6 8 11 15
7 16 3 10
5 16 7
4
5
6 18 19 10
5 11 12 17
16
15
15
2
```

For example in this graph we have 20 nodes and 24 edges. The next line suggests that node 1 is connected to node 2 , node 2 is connected to 1,8,9,20 and so on.We need to get this kind of format into metis.exe so when we first load the metis.html we'll generate this converted graph file. Here all the entities,activities and agents are stored in a way that they can be represented as a number.

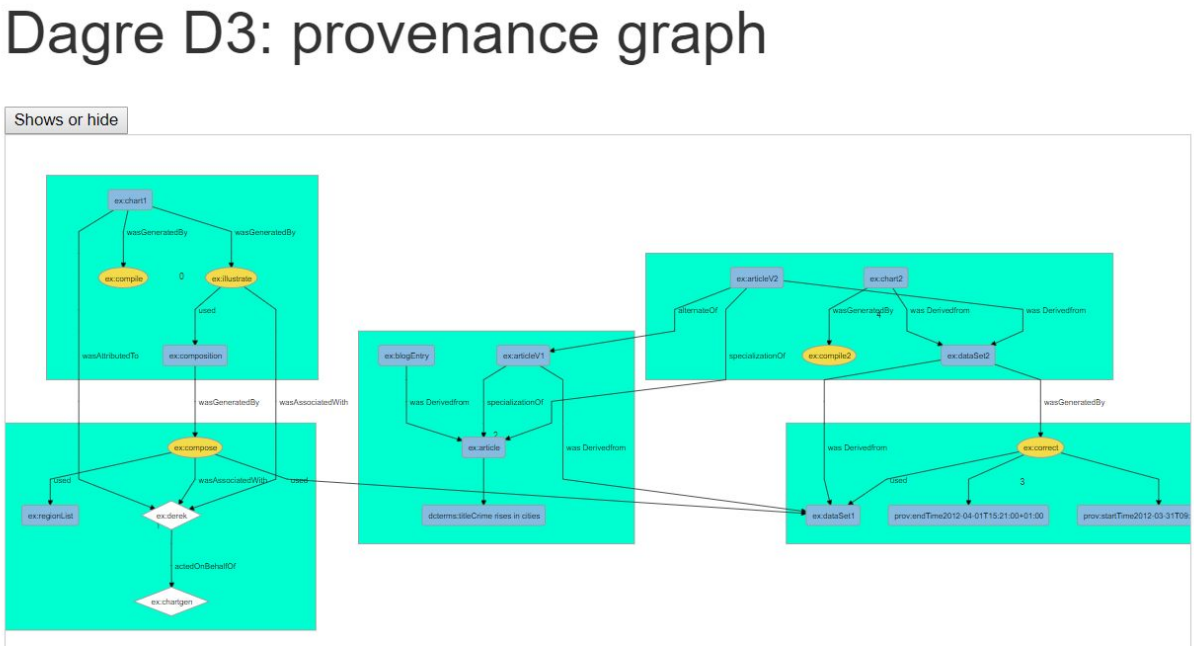


After we run metis on this graph file we'll get some information like edge cut and many other things which will look like this. This will generate a partitioned graph file. The format will show a group number on each line. Each line number is a node number and the number represented on that line is the group number for that node.

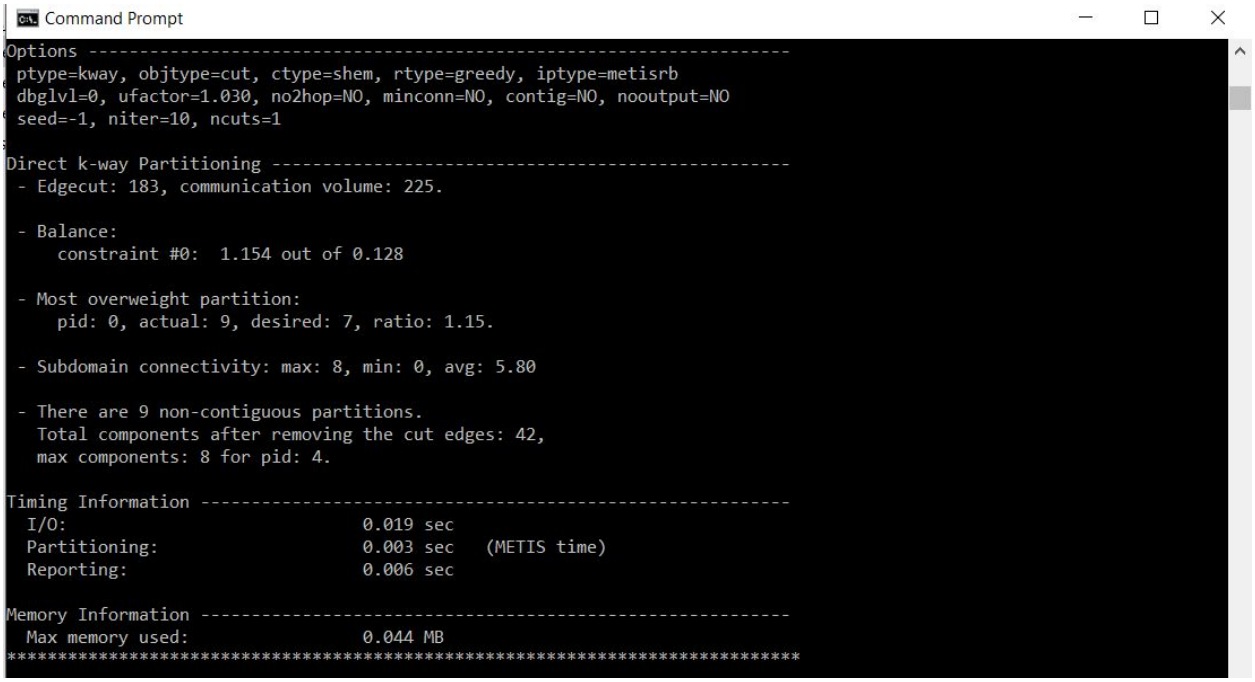
To make this file readable we need to add commas after each number and store them in some variable and store it as javascript file to read it in our code. It will look like this

```
var r = [2,
2,
1,
4,
0,
4,
0,
2,
4,
3,
1,
0,
4,
0,
3,
1,
1,
3,
3,
2,
]
```

After giving these file as an input in out metis.html file we'll get the partitioned graph looking like this after we refresh the same page again.



For graph 2 output will be



2.5 Greedy Partitioning

The greedy partitioning algorithm takes all the node in the input and starts putting the nodes into group based on their gain values and tries to avoid cycles. In this case all the groups will have same number of nodes in each groups. The main benefit of using it is that it avoids the cycles.

2.5.0 Greedy Partitioning Implementation

The algorithm for this partitioning is given in the paper "Acyclic Partitioning of Large Directed Acyclic Graphs".

To implement this algorithm we have to find the "GAIN" values for each node. We have a function named comp_gain which calculates the gain for each node. After getting the gain values we are sorting them according to it. After that we'll start putting them into the groups. By using this algorithm we'll get same number of nodes in each group.

```
function comp_gain(node,rev_part,i){
var gain = 0;
var u_pred = g.predecessors(allnodes[node]);
var u_succ = g.successors(allnodes[node]);

for(var a in u_succ){
    var succ_ind = allnodes.indexOf(u_succ[a]);
    if(rev_part.has(succ_ind)>=0 && rev_part.has(node) == rev_part.has(succ_ind)){
        gain = gain - 1;
    }
    else if(rev_part.has[succ_ind] == i){
        gain = gain + 1;
    }
}

for(var b in u_pred){
    var pred_ind = allnodes.indexOf(u_pred[b]);
    if(rev_part.has(pred_ind)>=0 && rev_part.has(node) == rev_part.has(pred_ind)){
        gain = gain - 1;
    }
    else if(rev_part.has(pred_ind)>=0 && rev_part.has(pred_ind) == i)
        gain = gain + 1;
    }
return gain;
}
```

For different number of partitions we get different number of nodes in each group and different number of edgecuts,

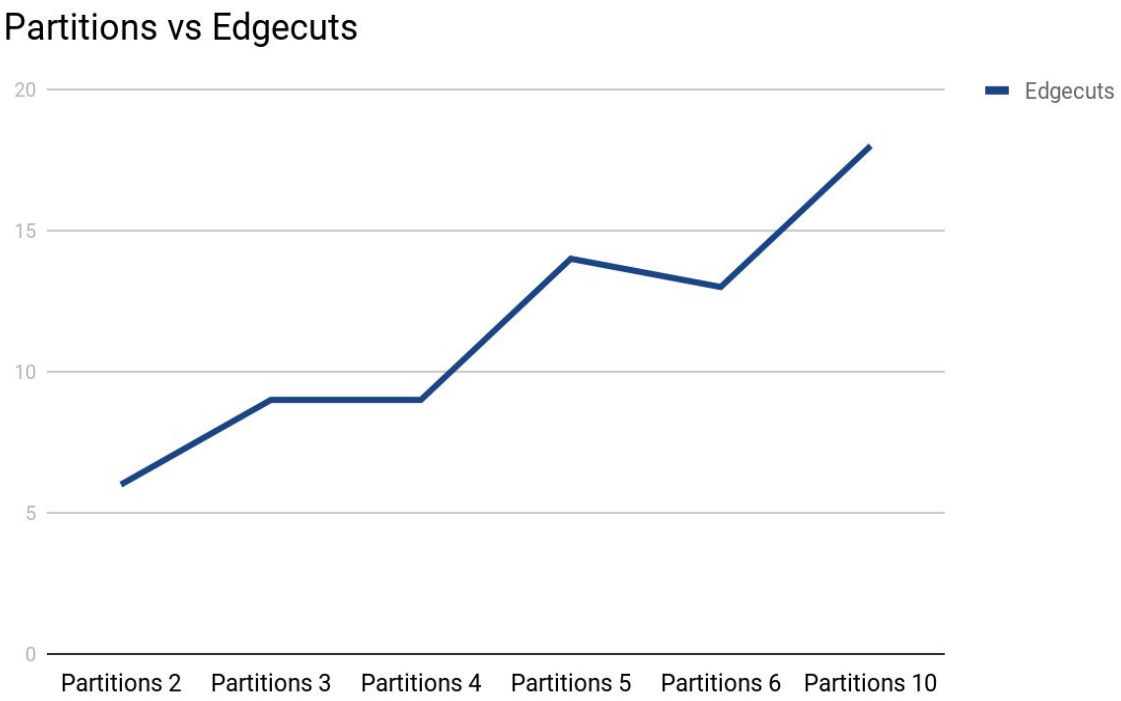
2.5.1 Greedy Partitioning Reinement Statistics

For graph article-prov1 with a different number of partitions we get this outputs.

According to this number of nodes in the groups and/or number of edge cuts we can select number of groups.

For graph 1

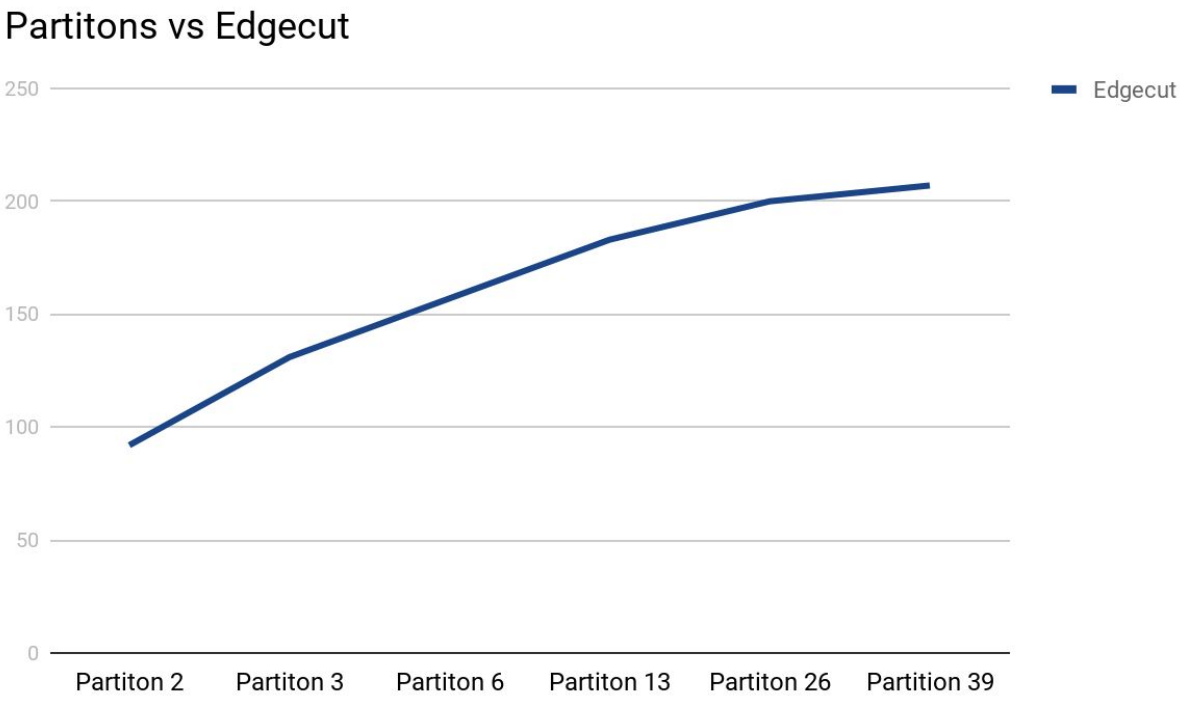
Graph number	Number of nodes	Number of edges	Number of partitions	Number of nodes in a group	Number of edgecuts
article-prov 1.json	20	24	2	9	6
article-prov 1.json	20	24	3	6	9
article-prov 1.json	20	24	4	5	9
article-prov 1.json	20	24	5	4	14
article-prov 1.json	20	24	6	3	13
article-prov 1.json	20	24	10	2	18



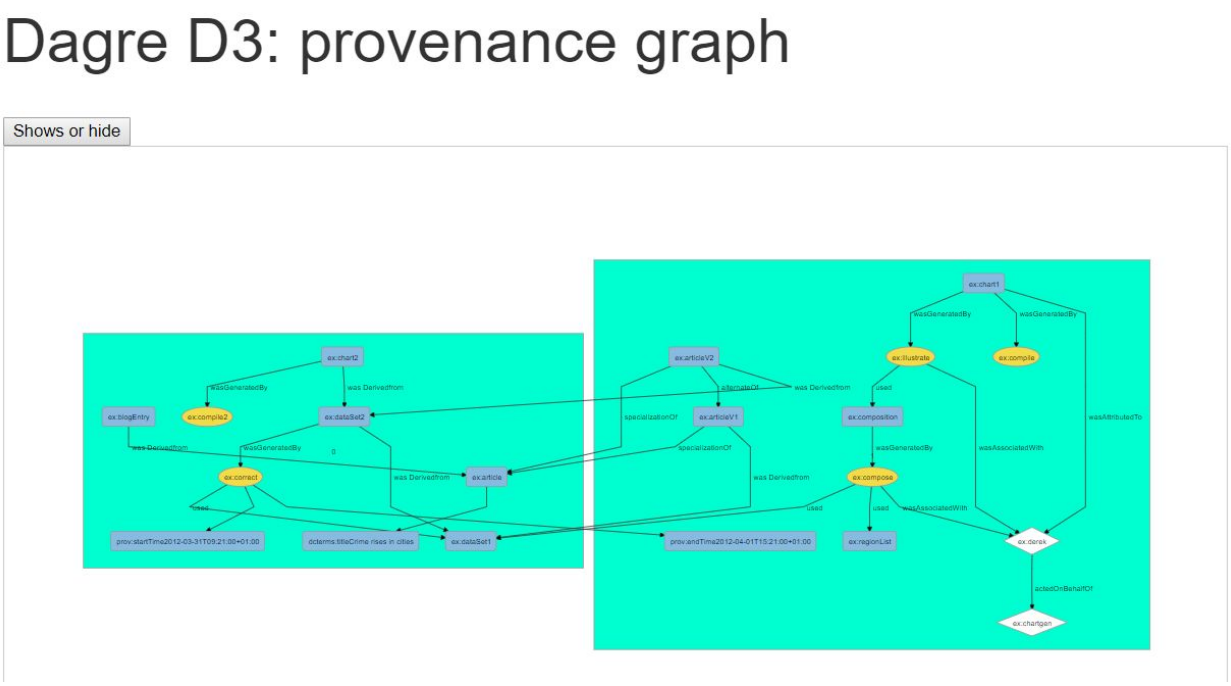
For graph 2

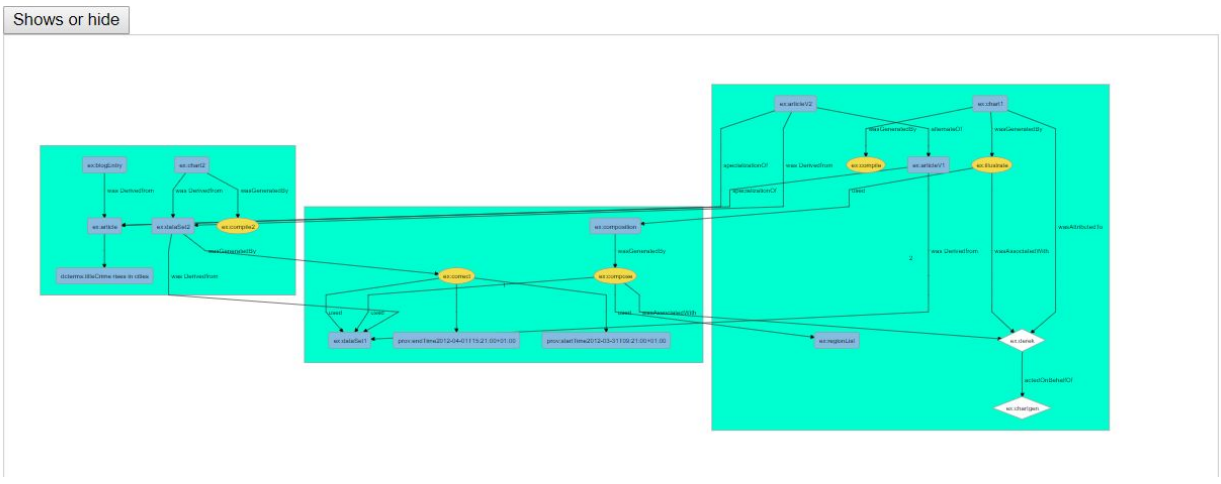
Graph number	Number of nodes	Number of edges	Number of partitions	Number of nodes in a group	Number of edgecuts
article-prov 2.json	78	219	2	35	92
article-prov 2.json	78	219	3	23	131
article-prov 2.json	78	219	6	11	157
article-prov 2.json	78	219	13	5	183

article-prov 2.json	78	219	26	2	200
article-prov 2.json	78	219	39	2	207



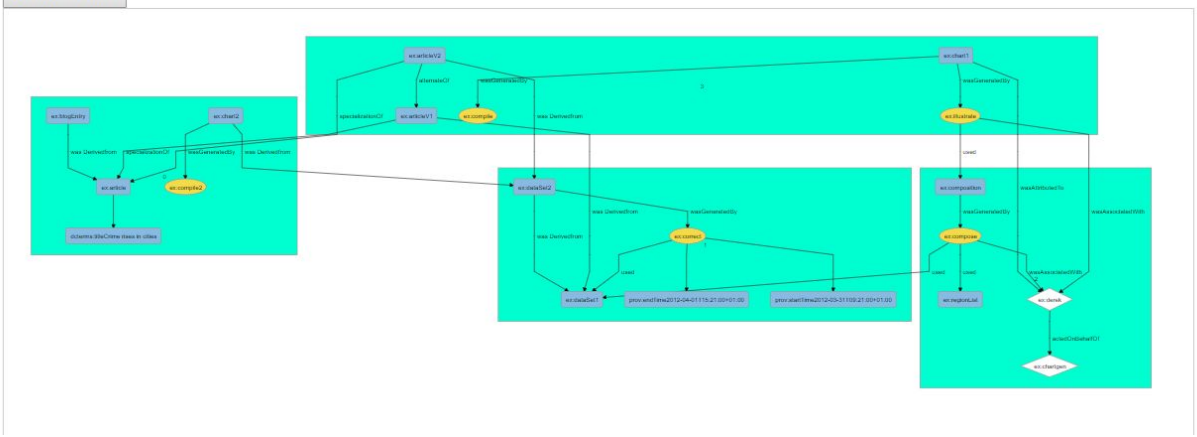
2.5.2 Output


Dagre D3: provenance graph
Shows or hide
ex:chart2
ex:blogEntry
ex:compile2
ex:dataSet2
ex:correct
ex:article
ex:dataSet1
dcterms:titleCrime rises in cities
prov:startTime2012-03-31T09:21:00+01:00
ex:chart1
ex:articleV2
ex:articleV1
ex:illustrate
ex:compile
ex:composition
ex:compose
ex:regionList
ex:derek
ex:chartgen
prov:endTime2012-04-01T15:21:00+01:00
wasGeneratedBy
wasDerivedFrom
alternateOf
specializationOf
used
wasAssociatedWith
wasAttributedTo
actedOnBehalfOf


Dagre D3: provenance graph
Shows or hide

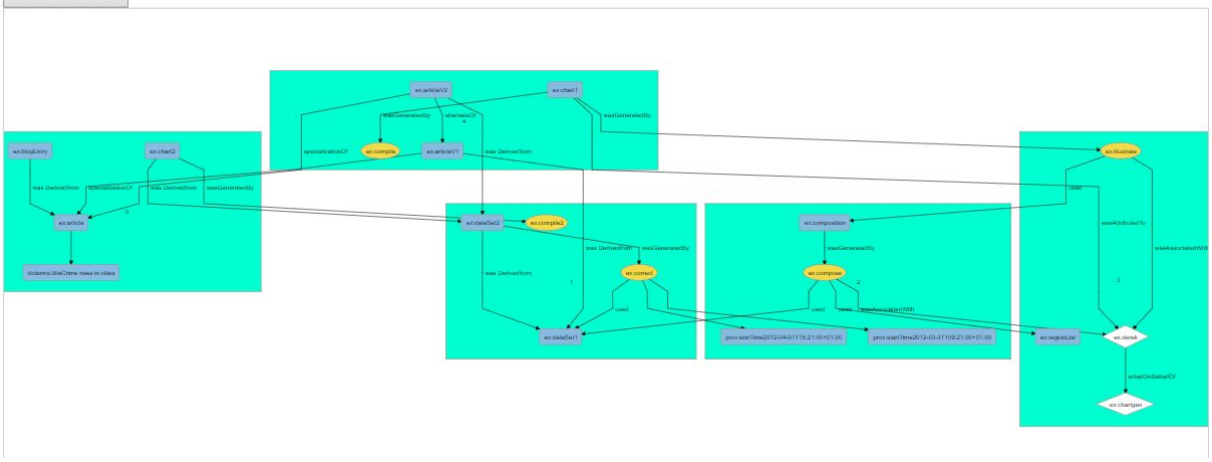
Dagre D3: provenance graph

Shows or hide



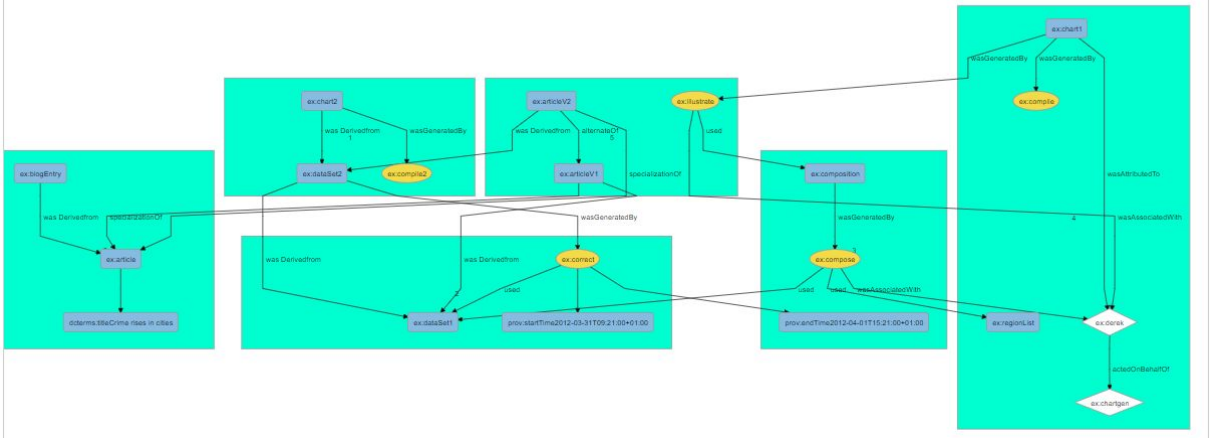
Dagre D3: provenance graph

Shows or hide



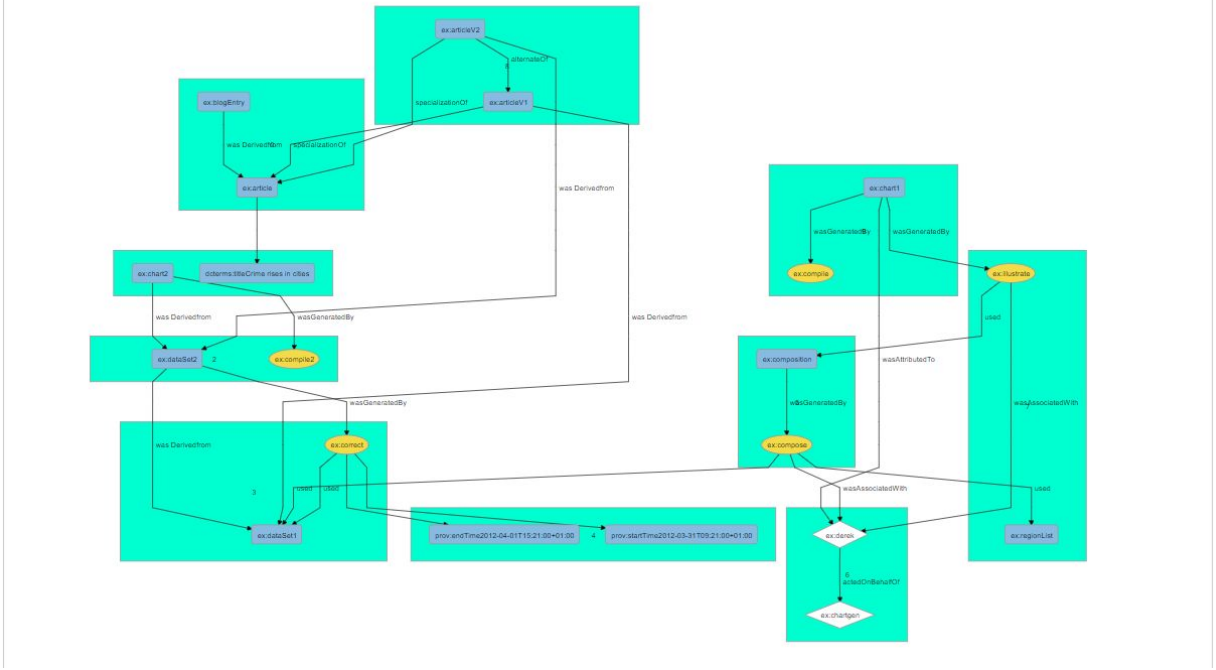
Dagre D3: provenance graph

Shows or hide



Dagre D3: provenance graph

Shows or hide



2.6 Greedy Partitioning Reinement

The refinement is very much needed after greedy partitioning. We don't want to have same number of nodes in all the groups. So this algorithm tries to use the boundary vertices and based on their parent and children nodes, tries to put them in another group. So for each node we are finding an update value of their new group. IT can be same and it can be a different group also.

By doing these we are reducing the dependencies of different groups among each other.

2.6.0 Greedy Partitioning Reinement

The algorithm for refinement is given in the same paper as greedy partitioning.

The refinement is very much needed after greedy partitioning. We don't want same number of nodes in all groups when we can minimize the edge cut by changing the group of few nodes. To implement this algorithm we are using UPDATE function. This update function takes the previous partitions as an input returns the new group for each node based on their min,max results. By doing refinement we can have less dependencies between groups and also less number of edge cuts. For this example as shown in the image shows that not all the groups have same number of nodes.

For this algorithm we need to find min and max values which are based on their predecessors and successors.

```
function Update(heap,moveto,gain1,u,part_parent){
    var max,min;
    var p = g.predecessors(u).length;
    if(p == 0){
        var max1 = part_parent[allnodes.indexOf(u)];
        max1 = max1 - 1;
        if(max1 > 1 ){
            max = max1;
        }
        else{
            max = 1;
        }
    }
    else{
        var predd = g.predecessors(u);
        var max1 =[];
        for(var v in predd){
            var indx = allnodes.indexOf(predd[v]);
            max1[v] = part_parent[indx];
        }
        max1.sort(function(a,b){return b-a});
        max = max1[0];
    }
    var q = g.successors(u).length;
    if(q == 0){
        var min1 =  part_parent[allnodes.indexOf(u)];
        min1 = min1 - 1;
        if(min1 > k){
            min = min1;
        }
        else{
            min = k;
        }
     }
    else{
        var succ = g.successors(u);
        var min1 =[];
        for(var v in succ){
            var indx = allnodes.indexOf(succ[v]);
            min1[v] = part_parent[indx];
        }
        min1.sort(function(a,b){return a-b})
        min = min1[0];
    }
```

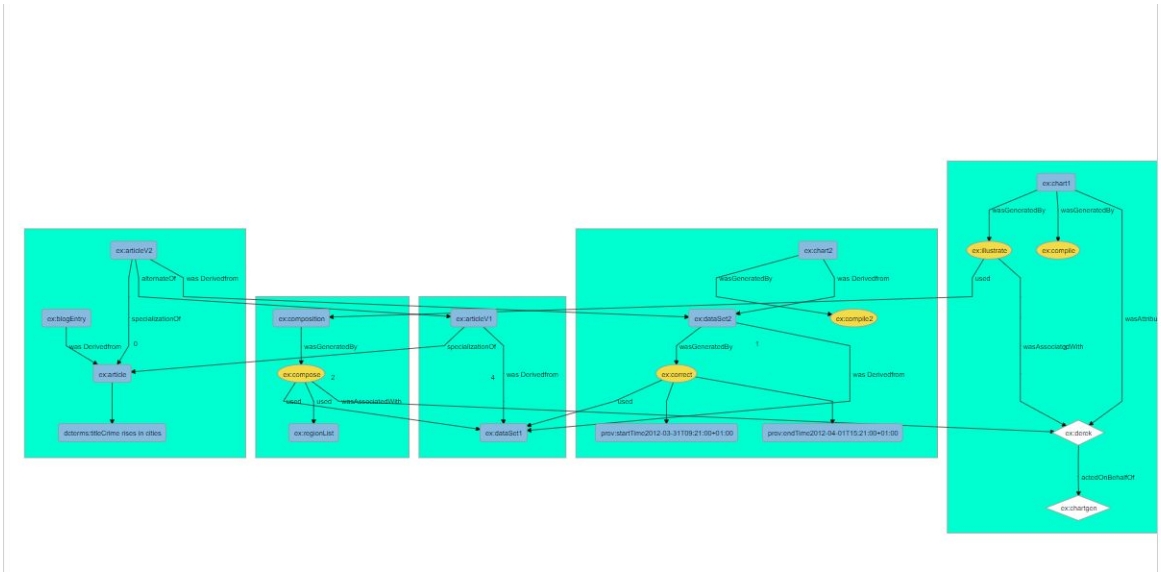
```
var part_indx = part_parent[allnodes.indexOf(u)];

if(max != part_indx && min == part_indx){
    heap.push(u);
    var ind = heap.indexOf(u)
    moveto[allnodes.indexOf(u)] = min;
    var gainiu = compgain_update(u,part_parent,max);
    //gain1[allnodes.indexOf(u)] = gainiu;
    gain1[ind]= gainiu;
    //console.log(gainiu);

}
if(min != part_indx && max == part_indx){
    heap.push(u);
    var ind = heap.indexOf(u)

    moveto[allnodes.indexOf(u)]  = max;
    //console.log(u)
    var gainiu = compgain_update(u,part_parent,min);
    //gain1[allnodes.indexOf(u)] = gainiu;
    gain1[ind]= gainiu;
    //console.log(moveto[u]);
}

if(min != part_indx  && max != part_indx){
    heap.push(u);
    var ind = heap.indexOf(u)

    var gain_1, gain_2;
    gain_1 = compgain_update(u,part_parent,min);
    gain_2 = compgain_update(u,part_parent,max)
    if(gain_1 > gain_2){
        moveto[allnodes.indexOf(u)]  = min;
        //gain1[allnodes.indexOf(u)]  = gain_1;
            gain1[ind]= gain_1;

    }
    else{
        moveto[allnodes.indexOf(u)]  = max;
        //gain1[allnodes.indexOf(u)]  = gain_2;
        gain1[ind]= gain_2;
    }
    //console.log(moveto[u]);
}
```

Until this part the algorithm is correct but

This algorithm is incomplete without one extra additional conditional. Here there needs to be a condition where min = max = part_index.

```
    if(max == part_indx && min == part_indx){
        heap.push(u);
        var ind = heap.indexOf(u)
        moveto[allnodes.indexOf(u)] = part_indx;
        var gainiu = compgain_update(u,part_parent,max);
        //gain1[allnodes.indexOf(u)] = gainiu;
        gain1[ind]= gainiu;
    }
```

This function will return us the updated values of gain based on their updated parent node. It will also give us the possible new parent (moveto).



2.6.1 Greedy Partitioning Reinement Statistics

*number of nodes can vary as per the refinement

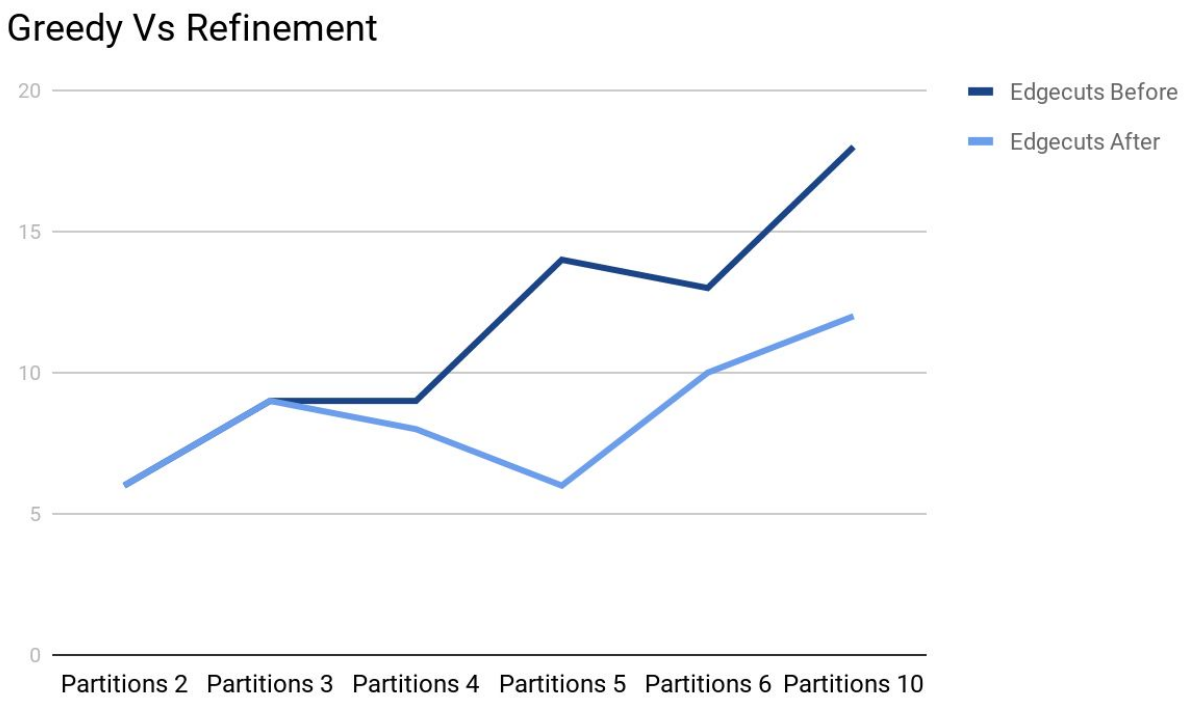
To find number of edgecuts we'll check for the successors of every node. If the parent of the node and the parent of the successors are different then there is an edgecut. By doing this for all the nodes in the graph we'll get the total number of egdecuts.

```
//inding number of edgeCuts
function edgexcut(allnodes){
    var edgecut1=0;
    for(var i in allnodes){
        var succ = g.successors(allnodes[i]);
        for(var j in succ){
            if(g.parent(succ[j]) != g.parent(allnodes[i])){
            edgecut1++;
        }
    }
}
 return edgecut1;

 }
```

For graph article-prov1 with a different number of partitions and also with refinement we get this outputs.

Graph number	Number of nodes	Number of edges	Number of partitions	Number of groups after refinement	Number of edgecuts after	Number of edgecuts before
article-prov1.json	20	24	2	2	6	6
article-prov1.json	20	24	3	6	9	9
article-prov1.json	20	24	4	4	8	9
article-prov1.json	20	24	5	4	6	14
article-prov1.json	20	24	6	3	10	13
article-prov1.json	20	24	10	7	12	18

For graph article-prov1 with a different number of partitions and also with refinement we get this outputs. Here is the comparison of greedy and greedy refinement. As we can see that by doing refinement we can reduce the number of number of edgecuts.



REFERENCES

1. https://github.com/cpettitt/dagre-d3
2. https://www.w3.org/TR/prov-primer/
3. https://github.com/dagrejs/graphlib/wiki/API-Reference
4. http://pages.cs.wisc.edu/~jignesh/publ/summarization.pdf
5. https://hal.inria.fr/hal-01672010/file/paper.pdf
6. http://glaros.dtc.umn.edu/gkhome/metis/metis/download