Efficiently Merging Graph Nodes

With Application to Cluster Analysis

Alex Ostrovsky

2007-04-20

An empirical analysis of undirected, weighted **graph data structures** is presented. The benchmark is agglomerative **cluster analysis**. A novel data structure is presented that is **faster than incidence lists and adjacency matrices**.

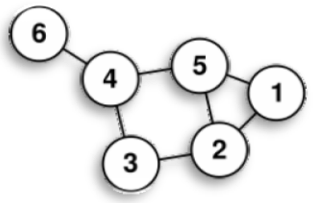
Efficiently Merging Graph Nodes with Application to Cluster Analysis

[Alex Ostrovsky – 2007]

Abstract

Undirected weighted graph data structures are benchmarked by performing cluster analysis. A novel graph data structure is presented that optimizes merging of nodes. Cluster analysis is shown to benefit significantly.

Introduction



Graph with six nodes, seven edges.

Graphs consist of nodes joined by edges. Nodes have data, and edges have weights. There are a many operations that make a graph, their performance varies with representation. Many problems can be expressed in terms of graphs, among them is cluster analysis. In cluster analysis items are grouped by similarity. Incidence lists, adjacency matrices, and FastGraph (a novel data structure) are benchmarked by performing cluster analysis. FastGraph is show to be the fastest and most memory efficient.

Outline

- The concept of a graph is defined it terms of the operations that can be performed on it.
- A rigorous definition of the node merge operation is given; this is the main operation of interest.
- An overview is given of overall concepts in graph data structures.
- Adjacency lists are discussed in detail.
- Pseudocode is given for the merge operation for an adjacency list.
- Adjacency matrices are discussed.
- Pseudocode and a graphical explanation are given for the merge operation of an adjacency matrix.
- The FastGraph data structure is discussed.
- A visual map of the FastGraph data structure is given.
- The Buffered FastGraph is discussed.
- Algorithmic efficiency is briefly touched on.
- Agglomerative cluster analysis is explained.
- 25 pages of benchmarks are presented.
- An outline of the source code is given.
- Selected source code is provided.

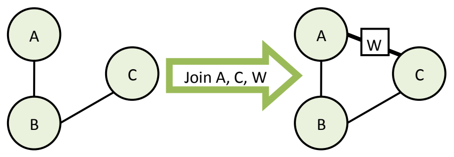
Graph Definition

A weighted undirected graph consists of:

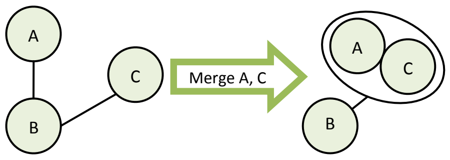
Nodes	• Each node has an associated data objects.
	• Node data can be merged with other data of the same type to form a union.
Edges	• Each edge joins a pair of nodes.
	• An edge connecting `A` to `B` equally connects `B` to `A` (i.e. they are undirected).
	• Edges have weights that can be viewed as real numbers.
	• An edge weight can be merged with another weight to form a union weight.
	• Nodes not joined by an edge are said to be connected by a special `null edge`.

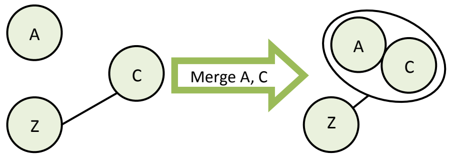
The following four graph operations are considered:

Join two nodes by an edge with the given weight. If the nodes are already joined by an existing edge then its weight is merged with the given one.



Merge two given nodes and return their union.
If they were connected by an edge, it will also be returned.





If nodes `A` and `C` are being merged to form `X` then:

- If node `B` has edges to both `A` and `C`, then the two edges are merged and made to join `X`.
- If node `Z` has an edge to only one of `A` or `B`, then it is merged with a special `null edge` and made to join `X`.
- All other nodes are unaffected.

Pseudocode for the merge operation is provided bellow.

Get two nodes joined by the edge with the lightest/heaviest weight.

Get an anti edge, which is an arbitrary pair of nodes that are *not* joined.

Rigorous Definition of Merge Operation

The node merging graph operation has the following pseudo code:

```
//-----------------------------------------------------------------
public Node_And_Edge merge( nodeData_A, nodeData_B )

    // used only as return value
    edge_AB = remove_edge_joining( nodeData_A, nodeData_B )
    union   = nodeData_A.merge_with( nodeData_B )

    for (related_node : nodes_incident_to_either(
                                nodeData_A, nodeData_B ))
        edge_RA = edge_joining( related_node, nodeData_A )
        edge_RB = edge_joining( related_node, nodeData_B )
        disjoin( related_node, nodeData_A )
        disjoin( related_node, nodeData_B )

        merged_edge = merge_nullable_edges( edge_RA, edge_RB )
        join( related_node, union, merged_edge )

    return new_Node_And_Edge(union, edge_AB )

//------------------------------------------------------------------
private EdgeWeight merge_nullable_edges(
                    edgeWeight_A, edgeWeight_B )

    if niether_are_null( edgeWeight_A, edgeWeight_B )
        return edgeWeight_A.merge_with( edgeWeight_B )

    if is_not_null( edgeWeight_A )
        return edgeWeight_A.mergeWith( edgeWeight_null )

    //if is_not_null( edgeWeight_B )
        return edgeWeight_B.mergeWith( edgeWeight_null )
```

Graph Data Structure Overview

There are two main families of graph data structures:

- In *adjacency lists*, each node tracks only other nodes that are joined to it by an edge.
- In *adjacency matrices*, each node tracks all other nodes whether or not an edge joins them.

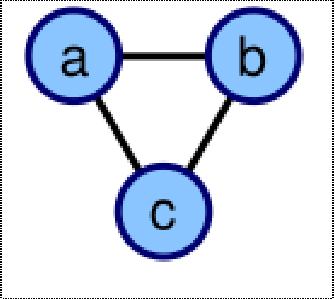
An important measure of graphs is their `density`, which `equals (number of used edges)`
`over`
`(potential edge count).`
In other words, a graph's density is the probability that two randomly selected nodes happen to be joined by an edge. For an undirected graph with `n` nodes, the potential edge count is `n*(n + 1)/2`.

If a graph has density greater than 0.5 it is called dense, otherwise it is called sparse. Adjacency lists are called a sparse data structures because they don't use memory for edges that are not there. Adjacency matrices on the other hand are considered a dense data structure.

Buffered FastGraph is a data structure that combines both sparse and dense properties. More specifically, an adjacency matrix is used as a buffer for *FastGraph*–a novel adjacency list data structure. Buffered FastGraph is faster than conventional adjacency lists or matrices according to my benchmarks.

Adjacency Lists

In an adjacency list data structure, each node keeps a list of all other nodes is has an edge to. The way a node's adjacency list is looked up, and how a list itself works vary widely. One reasonable adjacency list data structure suggested by (Rossum, 1998) uses a hash table used to associate each node with an array of adjacent nodes. Another possibility advocated by (Cormen, Leiserson, Rivest, & Stein) is an array indexed by node numbers pointing to a singly linked list of adjacent nodes.



a	adjacent to	b, c
b	adjacent to	a, c
c	adjacent to	a, b

One problem with adjacency lists is that there is no obvious place to store edge weights. For this reason some (Goodrich & Tamassia, 2002) advocate a more object oriented approach where each node stores a list of objects representing the edges incident to that node. In turn, each edge points back to the two nodes forming its endpoints. This is the approach considered in this paper, both in its original form and in a modified form called FastGraph.

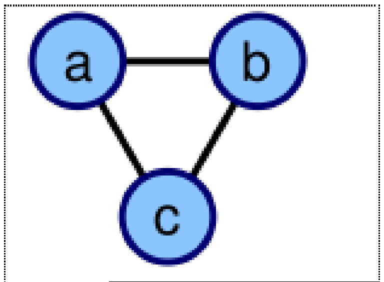
Pseudocode for Merge Nodes with Traditional Adjacency List

This pseudocode would apply to both (Rossum, 1998) and (Cormen, Leiserson, Rivest, & Stein).

```
//------------------------------------------------------------------
public Node_And_Edge merge( nodeData_A, nodeData_B )

      //map of edges that are incident to the node holding nodeData
      incidence_A = incidenceOf( nodeData_A )
      incidence_B = incidenceOf( nodeData_B )

      removeNode( nodeData_A )
      removeNode( nodeData_B )

      // used only as return value
      edge_AB = incidence_A.edge_joining( nodeData_B )
      union   = nodeData_A.merge_with( nodeData_B )

      incidence_A.remove( nodeData_B )
      incidence_B.remove( nodeData_A )

      for( edge : incidence_A )
          joined_to_A     = edge.other_endpoint( nodeData_A )
          from_A_to_union =
                  incidenceOf( joined_to_A ).remove( nodeData_A )
          from_B_to_union =
                  incidenceOf( joined_to_A ).remove( nodeData_B )
          incidence_B.remove( joined_to_A )

          // see above pseudocode for merge_nullable_edges
          union_edge = merge_nullable_edges(
                            from_A_to_union, from_B_to_union )
          join( joined_to_A, union, union_edge )

      for( edge : incidence_B )
          joined_to_B = edge.other_endpoint( nodeData_A )
          union_edge  = merge_nullable_edges( edge, null )

          join( joined_to_B, union, union_edge )

      return new_Node_And_Edge( union, edge_AB )
```

Adjacency Matrices

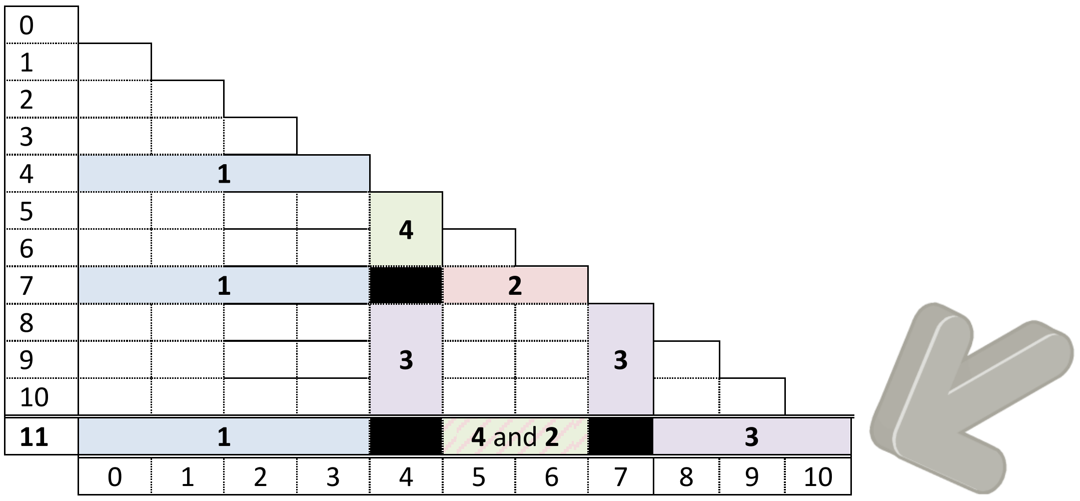
An adjacency matrix is based on a two dimensional array (aka. matrix). In order to map nodes to array indices all nodes in a graph are labeled with consecutive natural numbers. If two nodes are joined by an edge, then its weight is stored in the matrix at the intersection of their indices.



	a [0]	**b** [1]	**c** [2]
a [0]			
b [1]	edge		
c [2]	edge	edge	

Since we are dealing with undirected graphs, only half of the matrix needs to be used (see figure to the right). Given two indices `x` and `y` the associated edge is at `matrix[max(x, y)][min(x, y)]`.

When merging two nodes, a new row is added to the matrix to represent their union. The union row it is populated by merging the edges of the two given nodes. There are four distinct cases that can be considered, for an example when merging nodes 4 and 7 in a 10 node matrix:



Union row is inserted here after the 10th row.

1) Vertical overlap: populates the start of union row.
2) Vertical high index overshoot: overlaps with (4) to populate the middle.
3) Horizontal overlap: populates the end of the union row.
4) Horizontal low index overshoot: collaborates with (2).

Refer to the following two pages for pseudocode for the merge nodes operation with the adjacency matrix data structure.

```
//-----------------------------------------------------------------
public Node_And_Edge merge( nodeData_A, nodeData_B )

      index_A = indexOf( nodeData_A )
      index_B = indexOf( nodeData_B )

      if index_A < indexB
          return merge( index_A, nodeData_A,
                        index_B, nodeData_B )
      else
          return merge( index_B, nodeData_B,
                        index_A, nodeData_A )

//------------------------------------------------------------------
private Node_And_Edge merge(loIndex, loData,
                            hiIndex, hiData )

    union      = loData.merge_with( hiData )
    xWeight    = matrix[hiIndex][loIndex]
    unionIndex = add(union)

    horizontal_overlap(loIndex,    loData,
                       hiIndex,    hiData,
                       unionIndex, union )

    horizontal_hi_overshoot(loIndex,
                            hiIndex,    hiData,
                            unionIndex, union )

    vertical_overlap(loIndex,    loData,
                     hiIndex,    hiData,
                     unionIndex, union )

    vertical_lo_overshoot(loIndex,    loData,
                          hiIndex,
                          unionIndex, union )

    return new_Node_And_Edge( union, xWeight )

//------------------------------------------------------------------
private void horizontal_overlap(loIndex,    loData,
                                hiIndex,    hiData,
                                unionIndex, union )

    loLows    = matrix[ loIndex    ]
    hiLows    = matrix[ hiIndex    ]
    unionLows = matrix[ unionIndex ]

    for( i : 0 ... (loIndex - 1) )
        unionLows[i] = merge_nullable_edges( loLows[i], hiLows[i] )

    matrix[loIndex][0 ... (loIndex - 1)] = null
```

```
//------------------------------------------------------------------
private void horizontal_hi_overshoot(loIndex,
                                     hiIndex,    hiData,
                                     unionIndex, union )

    hiLows    = matrix[ hiIndex    ]
    unionLows = matrix[ unionIndex ]

    for( i : (loIndex + 1) ... (hiIndex - 1) )
        unionLows[i] = merge_nullable_edges( hiLows[i], null )

    matrix[hiIndex][0 ... (hiIndex - 1)] = null

//------------------------------------------------------------------
private void vertical_overlap(loIndex,    loData,
                              hiIndex,    hiData,
                              unionIndex, union )

    unionLows = matrix[ unionIndex ]

    for( i : (hiIndex + 1) ... (unionIndex - 1) )
        subMatrix = matrix[i]

        unionLows[i] = merge_nullable_edges( subMatrix[loIndex],
                                             subMatrix[hiIndex] )
        subMatrix[loIndex] = null
        subMatrix[hiIndex] = null

//------------------------------------------------------------------
private void vertical_lo_overshoot(loIndex,    loData,
                                   hiIndex,
                                   unionIndex, union )

    unionLows = matrix[ unionIndex ]

    for( i : (loIndex + 1) ... (hiIndex - 1) )
        overlap = matrix[i][loIndex]

        if( overlap is not null )
            unionLows[i] = merge_nullable_edges( overlap,
                                                 unionLows[i] )
            matrix[i][loIndex] = null
```

FastGraph Representation

FastGraph is an adjacency list variant specifically designed for efficiently merging nodes.

- Uses a hash table to look up nodes from their data.
- Nodes each have data and a label.
- There are two edge incidence lists per node.
    - Low-high incidence: stores edges to nodes with higher label.
    - High-low incidence: stores edges to nodes with lowed label.
- In both incidence lists of a node `n`, each edge's *major* endpoint is `n`. The other endpoint is called *minor*.
- An incidence list is a bidirectional linked list of edges stored in ascending order of minor endpoint labels.
- Each edge appears in exactly two incidence lists, one low-high incidence, and one high-low incidence.
- Edges store their previous and next links in both of the high-low, and low-high incidence lists that they appear in.

For maximum efficiency, the `O(n)` bucket sort is used to keep track of edge weights so that the next heaviest edge can be looked up. This is actually true about all tested implementations; however this step has been omitted from the pseudocode to the sake of clarity.

Example set of edges when looked at in low-high order:
(0, 1) (0, 2) (0, 5) (1, 2) (1, 3) (2, 5) (4, 5)

Here we have four distinct low-high incidence objects, where the number before the bar represents the major index.
(0 | 1, 2, 5)   (1 | 2, 3)   (2 | 5)   (4 | 5)

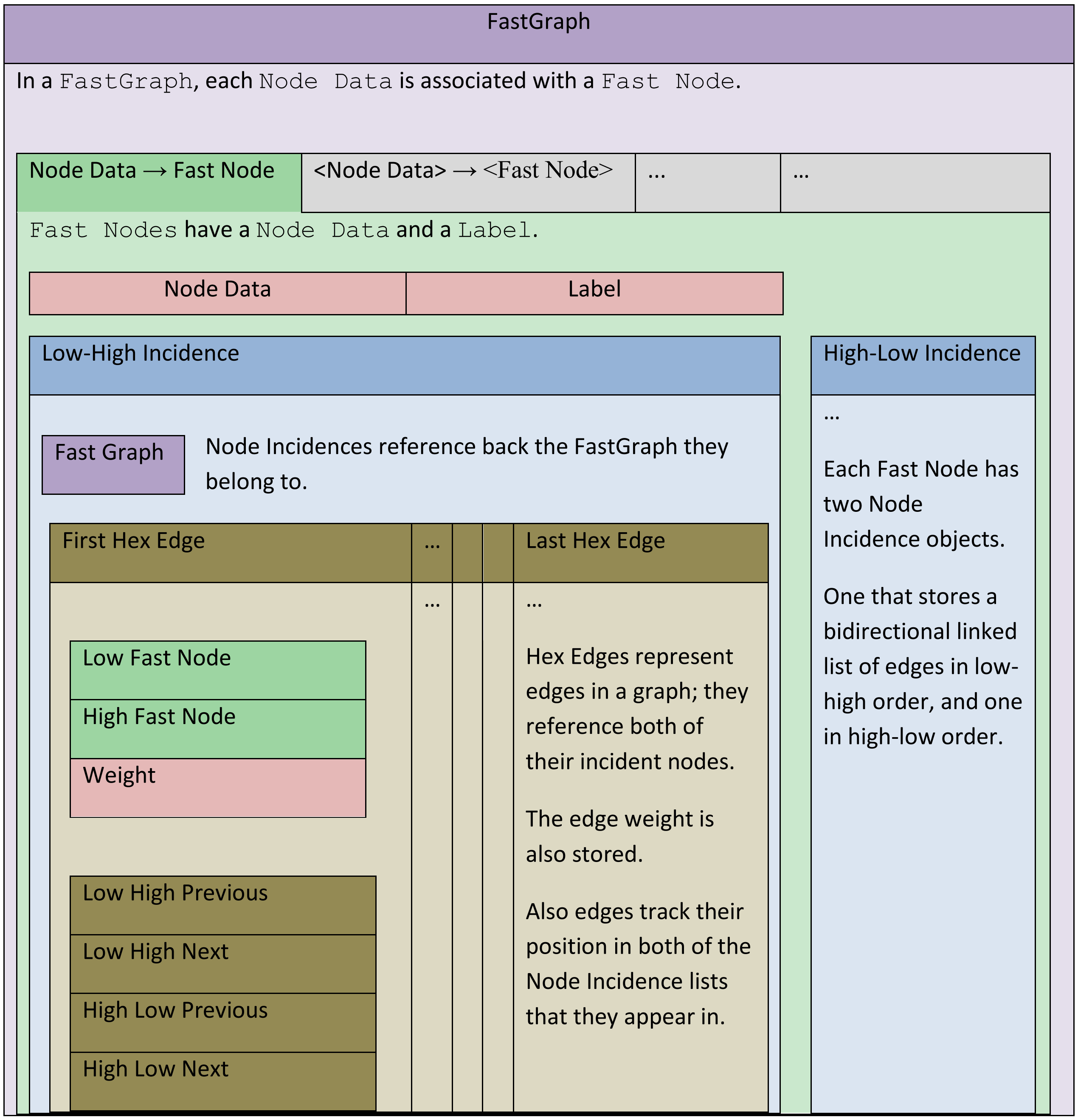
From the point of view of high-low incidences, the same edges are grouped like this:
(1 | 0)   (2 | 0, 1)   (3 | 1)   (5 | 0, 2, 4)

Please refer to the annotated table below for a visual expose of the FastGraph data structure.

When two nodes are merged, their incidence lists are traversed with an erasing curser. Since edges in the incidence lists are sorted, the erasing cursers can be efficiently merged into a union incidence list.

The code that dose the merging is rather complex so I will not provide any pseudocode, rather I encourage you to look at the actual source code that is provided in Appendix B.

Visual Summary of FastGraph Data Structure



In a `FastGraph`, each `Node Data` is associated with a `Fast Node`.

`Fast Nodes` have a `Node Data` and a `Label`.

Node Incidences reference back the FastGraph they belong to.

Hex Edges represent edges in a graph; they reference both of their incident nodes.

The edge weight is also stored.

Also edges track their position in both of the Node Incidence lists that they appear in.

Each Fast Node has two Node Incidence objects.

One that stores a bidirectional linked list of edges in low-high order, and one in high-low order.

Matrix Buffered FastGraph

The best performing graph variant was an adjacency matrix that turned into a FastGraph as soon as node merging began. Please refer to the benchmarks in Appendix A for more details.

Algorithmic Efficiency

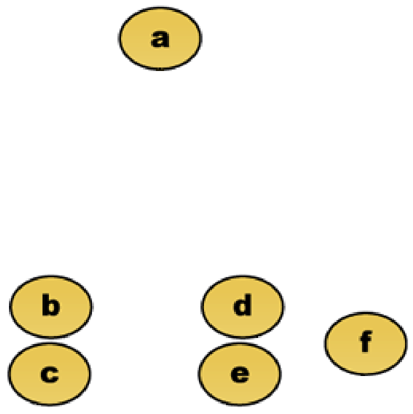
I attempted to derive memory and runtime efficiency formulas, but unfortunately the evidence totally contradicted my calculations. For example, it turns out that an adjacency matrix is almost always more memory efficient than an adjacency list.
See Appendix B for more details.

Application to Cluster Analysis

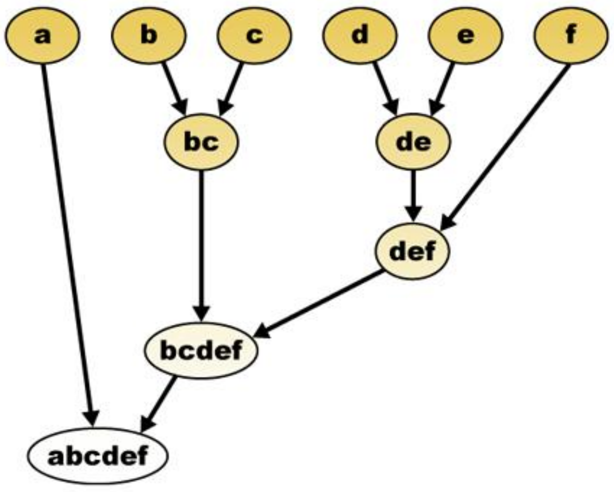
A common problem in the sciences is how to organize observed data into meaningful structures. Cluster analysis aims at sorting different items into groups such that difference within a group is minimal, and difference between groups is maximal. Cluster analysis can be used to discover structures in data without providing any explanation for their existence.

There are many types of clustering algorithms, the most general one being agglomerative cluster analysis. The output of clustering is a hierarchy of items. The traditional representation of this hierarchy is a tree called a dendrogram. It has individual items at one end and a single cluster containing every item at the other.

Agglomerative cluster analysis starts with each item being a cluster, then the next closest pair of clusters (as determined by edge weight) are repeatedly merged until only one cluster is left.



Data to be clustered.

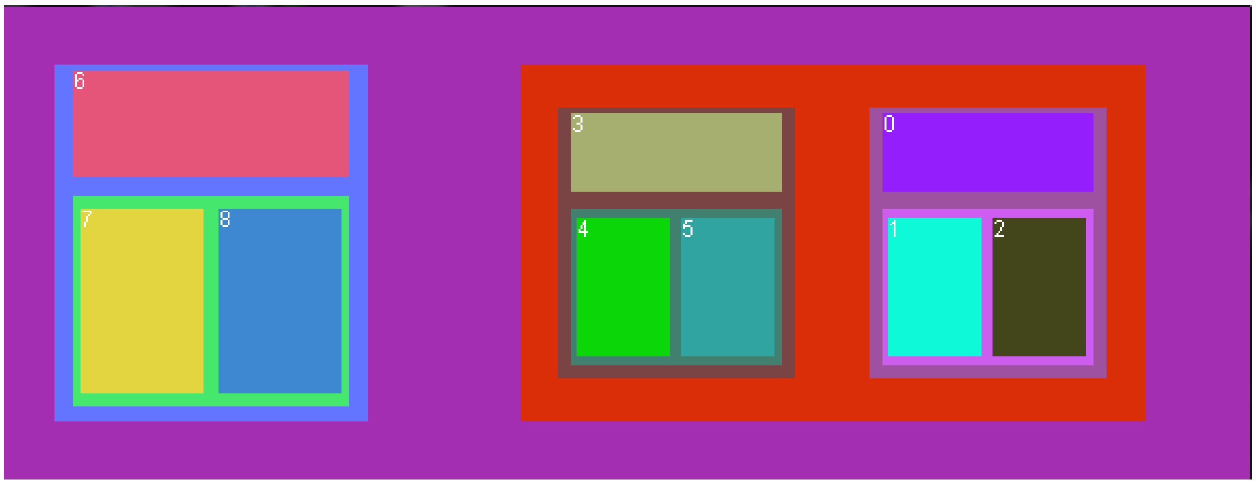


Dendrogram.

Cluster analysis is used as a benchmark. For example, given the non Euclidean relations:

- (0, 1) (0, 2) (1, 2)
- (3, 4) (3, 5) (4, 5)
- (6, 7) (6, 8) (7, 8)

The visual output that the benchmark gives is:



Works Cited

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. *Introduction to Algorithms, Second Edition.* MIT Press and McGraw-Hill.

Goodrich, M. T., & Tamassia, R. (2002). *Algorithm Design: Foundations, Analysis and Internet Examples.* John Wiley & Sons.

Rossum, G. v. (1998). *Python Patterns — Implementing Graphs.* Retrieved from Python Software Foundation: http://www.python.org/doc/essays/graphs/