ABSTRACT
Comprehensible high level representations of software development processes are often not available. By using a clustering algorithm, software development tasks can be represented in a more comprehensible way when using a new diagram style which represents developers as well as the tasks they are performing.

Integration in the TESNA project has succeeded and showed that the algorithms work for inputs from real projects. In this way, a diagram style to comprehensibly display software development processes has been created.

Keywords
DSM, design structure matrix, diagram, software development, development visualization, clustering algorithm, TESNA

1. INTRODUCTION
Software development processes often involve a number of developers working on hundreds of software modules. It is therefore difficult to keep an overview of who is, or should be, working on which part of the system being developed. This paper describes the results of research aimed at the comprehensible representation of software development tasks.

2. DEFINITIONS
This chapter gives definitions for some concepts used in this paper to provide background information to the reader unfamiliar with those concepts.

2.1 The DSM
A Dependency Structure Matrix, or DSM, can be used to represent task interdependencies within a business process [EPP01]. It can provide information about which tasks depend on which, can be used to identify problems in the task sequence and can give ideas on how to cluster tasks. Figure 1 shows an example of a simple DSM.

The letters A-E, on both axis of the matrix, represent tasks. Tasks in a business process are executed in the order A-E. An ‘x’ in location (a,b) of the matrix means that the task of row a depends on the task in column b. Dependencies below the gray diagonal represent ‘feed forward information’, while tasks above the diagonal represent feedback, for example, task E gives feedback on task C. In this example, tasks A and B depend on each other, and should therefore be clustered (performed simultaneously).

![Figure 1. Example of a DSM](image)

2.2 DSM in Software
Dependency Structure Matrices can also be applied to software systems [SAN05]. When representing a software system using a DSM, the tasks along the axis are replaced by modules of a software system (i.e. classes or packages). It is a very common practice to have a modular software design and this is also considered very valuable [SUL01]. The dependencies in the DSM indicate which module depends on which (i.e. by function calls and/or variable references).

Also, a matrix can be used to represent which developer is working on which software system module [AMR07]. In this case, the software modules are listed along one axis, while the members of the development team are listed along the other. Dependencies indicate who is working on which module(s).

2.3 Software Development Process
Generally, the process of software development includes several phases, i.e. phases for requirements engineering, global and detailed design, implementation, testing and maintenance [LAU02]. In this paper, however, concerning the subject of this research, the focus lies on the implementation phase. By ‘software development process’, we mean the actual programming of the software. Also, the implementation is assumed to be performed by a relatively small software development team, instead of hundreds of people usually working on a large open source project. [DEM02]

3. RESEARCH SUBJECT AND ASPECTS
Technical dependencies among software modules create social dependencies among software developers [SOU04]. The combination of information about software module interdependencies and information about who is working on which software module can give insight in the structure of the software development process.

However, the amount of modules in software systems can become very large. For example, software systems like the
The number of potentially relevant articles found in the literature was very large. Judging on the abstracts, a lot of these turned out to be too far off-topic to be useful in this research. Still, a significant number of articles seemed to be worthwhile to read to determine their relevance in this research. The findings in the literature study are presented here.

4.2 Diagrams for Programmers

First and most obvious, a lot of diagrams exist to display the structure of a software system. The widely-used Unified Modeling Language (UML) consists of several diagram styles, i.e. class diagrams, use case diagrams and sequence diagrams. Aside from software architecture representations, these diagrams are also often used for requirements specification [FOW03][LAU02]. Although UML diagrams can contain a lot of information about the structure, state of a software system, or the interaction with users, it does not contain diagram styles concerning developer - software module dependencies.

A significant amount of research has been done on the way the software development environment and the software being developed itself affect each other. See, for example, [PER88],[STI89]. This environment can also contain software tools that support software development (in this case, the tool is seen as the ‘environment’. For example, Netbeans and Eclipse are software developments environments in this context). These tools are expected to evolve into large systems that are more integrated with rapidly changing external environments and use more formal methodologies than before, which will make them better adapted and more effective [OSS00]. Although these tools can significantly aid the programmer in coding and collaborating with team members, the literature concerning these tools remains very silent on visual representation issues.

A lot of tools exist that can generate diagrams from source code, and also, software development tools exist that produce ‘real time’ diagrams from the source code of the software being developed (generate some source code from diagrams being drawn) [MAT02][BUS91][GUE04]. These applications are capable of producing detailed diagrams for parts of a software system being developed, or detailed diagrams for the systems as a whole. In the later case, however, no automated clustering is done, which can result in diagrams of unmanageable sizes.

All of the results that have been presented in this paragraph so far mainly deal with representing details of a software system, and therefore support programmers. The diagrams mentioned therefore are not suitable for high level representation of software development processes.

4.3 Software Development Representation

The results presented earlier imply that in software development research no or little attention is being paid to the representation of development processes for larger software systems in a single (or few) high-level diagram(s) in a comprehensible way. This is, however, not entirely true. Examples of research in this area include the following:

4.3.1 3D Visualization

A three-dimensional graphical representation concept for software visualization has been used to develop the sv3D (Source Viewer 3D) framework, which can be used to visualize large software systems [MAR03].

The method is based on the Seesoft representation [EIC92][STE93]. Goal of this project is to support software engineers in tasks at the level of software modules. Figure 3 shows an example of output from this tool. This representation
does not use DSMs or a similar dependency representation as input but works directly on source files.

Figure 3. SV3D Visualization example. Each cylinder represents a line of code (source: [MAR03])

It operates at a lower level than the representation of which this research is the subject (it is said to be ‘line oriented’). A lesson that can be learned from this project, however, is that the selection of the metaphor to use when representing something needs sufficient attention, as it may make the difference between an easily understandable diagram and one that is difficult to comprehend [DIX04].

4.3.2 Knowing what is going on
Another interesting aspect of the matter at hand which has been researched is awareness, or ‘knowing what is going on’, in collaborative software development activities. In a global development environment, coordination is very important and the visualization of development processes can aid coordinators in their tasks. A very relevant paper which addresses this issue is [STO05]. Using five dimensions (intent, information, presentation, interaction and effectiveness), a number of existing tools concerning (among other things) software development visualization are evaluated. These include SeeSoft, VRCS, Tukan, and Xia/Creole, ADVIZOR, Palantir, softChange, evolution Matrix, Augur, and Spectograph. The main idea of these tools is to synthesize available information on a software development process, mainly to create awareness for people involved in this process, so that they know what is going on. Still, however, these tools are not meant for high level representation, as they are meant to aid the programmer in his or her work on code and therefore mainly display low-level information.

4.4 Other Examples
The tools and diagrams mentioned so far are only examples of what has been found in his literature research. Papers have been written on lots of other aspects of software visualization, and it is impossible to show all research results here. Some examples include the high-level visualization of the execution of software, mainly meant for maintenance tasks [WAL98], software release history representation using three-dimensional colored representations [GAL99]. In general, software development tools mainly focus on supporting technical work. This has also been concluded by [HAL06], which itself poses an interesting method for change management visualization. Others have also researched the visualization of the evolution of software, see for example [VOI06]. Actually, symposia on the single subject of software visualization are organized [SVS06]. There are tools that also take developers into account when displaying a software development prices. For example, a tool called Augur exists, which creates representations of both software modules and development activities and explore their relations [FRO04]. These visualizations are designed to aid the programmers. Attempts are made to fill the gap between diagrams for developers and high level representations [MUR95].

4.5 Conclusion
From the conducted literature research it can be concluded that an enormous amount of research on various aspects of software visualization has been done. Still, however, the concept of global overview of development processes, involving software modules as well as developers, and their relationships, has not been encountered. The research described in this paper fits in that hole.

5. CLUSTERING ALGORITHM
The purpose of a clustering algorithm in this context is to reduce the size of the dependency matrix of software modules to a size in which it is usable as input for creating a diagram. Initially, a literature research was planned to determine an optimal clustering algorithm to use in this research. However, because the fact that research concerning this issue had already been done at the IS&CM department of the University of Twente which supervised this project, the choice has been made to use the clustering method found in that research (which itself claims that not many alternatives for this specific purpose are available), and implement it in a modular, adaptable way to allow optimizations and integration with the TESNA tool.

5.1 Algorithm Origin
The clustering algorithm originates from a Master Thesis written at the Massachusetts Institute of Technology by C.I.G. Fernandez [FER98], which itself is based on an algorithm developed by J. Iduica [IDJ95]. This work defines a general method for clustering data in DSMs. It can be applied to various kinds of information that can be contained in DSMs, among which software systems.

5.2 Algorithm Mathematics
The algorithm attempts to cluster information by minimizing the ‘total coordination cost’, which is the sum of all the coordination costs for individual tasks. The coordination costs of a single task, where in this case the task is the development of one software module from a system, are defined as follows:

\[
CC(i) = \sum_{j=1}^{n} (SDM(i,j) + SDM(j,i)) \times \text{size}(i,j)^2
\]

Equation 1: Cluster cost function

In this formula \(CC(i)\) are the coordination costs of task \(i\), \(SDM(i,j)\) is the value at coordinates \((i,j)\) of the Software Dependency Matrix, and \(\text{size}(i,j)\) is the total number of software modules \((=n)\) if \(i\) and \(j\) do not belong to the same cluster, or the size of the cluster \(i\) and \(j\) are both in otherwise.

In the mentioned paper [FER98], two approaches for the stated minimization problem are presented. The first one is a stochastic approach, which randomly selects a module and assigns it to the cluster it best fits in (determined by a bidding process), and basically repeats this procedure until the total coordination cost no longer declines. It does not necessarily find an optimal solution, but has a polynomial space/time complexity. The second one is a deterministic mathematical approach which basically tries all possibilities. Due to the
exponential complexity of this algorithm, it is not suitable for large input sizes [BAA00]. In general, any deterministic algorithm which finds an optimal solution to this particular problem will have an exponential time complexity: assuming a software system of \( M \) modules and a fixed number of clusters \( C \), with \( N \) clustering possibilities (so \( N=C^M \)), adding one module to the system creates \( NC \) clustering possibilities since the new module can be placed in any of the clusters. In the same way, adding \( i \) modules causes \( N^C \) clustering possibilities. For this reason, a deterministic approach is not usable in this research and the stochastic approach to data clustering has been used in this research.

5.3 Algorithm Description

The algorithm that is the result of this research will be described by first explaining the original algorithm, and then specifying optimizations and parameters. The most important parts of the source code can be found in appendix A.

Fernandez’ stochastic algorithm initializes by assigning each software module to its own cluster, so that the total number of clusters equals the number of modules. Then the total costs of this initial ‘clustering’ are calculated by the formula mentioned earlier.

The algorithm then enters its main loop. It selects a random software module and assigns it to the cluster which makes the highest bid on it, where the bid is calculated upon the interdependency between the cluster and the module: the higher the interdependency, the higher the bid (the exact formula is
\[
\text{highest bid on it, where the bid is calculated upon the interdependency between the cluster and the module: the higher the interdependency, the higher the bid (the exact formula is)}
\]

\[
\text{The initial cost value is calculated using Equation 1}
\]

\[
\text{int cost = calculate initial total cost;}
\]

\[
\text{int failed_attempts = 0;}
\]

\[
\text{Do {}
\]

\[
\text{m = a random module from inputDSM;}
\]

\[
\text{For every element c from clusters {}
\]

\[
\text{calculate bid of c for m;}
\]

\[
\text{}}
\]

\[
\text{cmax = highest bidding cluster;}
\]

\[
\text{best_bid = bid from cmax;}
\]

\[
\text{If (best_bid > 0) {}
\]

\[
\text{Reassign m to cmax;}
\]

\[
\text{For every element c from clusters {}
\]

\[
\text{if (c is empty) remove c;}
\]

\[
\text{}}
\]

\[
\text{cost = cost – best_bid;}
\]

\[
\text{failed_attempts = 0;}
\]

\[
\text{}}
\]

\[
\text{failed_attempts++;}
\]

\[
\text{while (failed_attempts <= FAILED_LIMIT) {}
\]

\[
\text{Return clusters;}
\]

5.4 Testing and Optimization

The described algorithm has been implemented in the testing tool written in the context of this research. It involves a number of parameters that tweak the clustering results and the speed of the algorithm. The implementation (with some javadoc comment) can be found in the Java source code file MacCormackClusterer.java. See the chapter ‘diagram development tool’ for details.

5.4.1 Initial test results

To provide input to the algorithm, a matrix is generated from a randomly generated graph using the Java Universal Network/Graph framework [JUN07]. The reason for this is that the same framework is used by the TESNA tool [TES07], in which the results of this research are to be implemented, so compatibility has to be guaranteed. The test can be repeated as often as needed with user-entered input size. Test results show that, before any optimization has taken place, the clustering of a matrix with 100 software modules takes approximately 14 seconds on average. Due to the non-deterministic nature of the algorithm the execution time varies between different attempts. In the same way, clustering 50 modules takes about 1 second, and clustering 200 modules takes 200 seconds. This implies a more or less linear time complexity in which an increase of the software system by a factor 2 causes an increase in calculation time of a factor 14. Analysis of the algorithm’s source code by [BAA00] confirms that the time complexity is in \( \Omega(n) \) and in \( O(n^2) \), where \( n \) is the number of software modules. The fact that the relationship is quadratic in the worst case is very good, since this makes the algorithm (potentially) usable for large input sizes, which is needed. The constant factor 14, however, is still quite large in this initial test. For this reason, options to speed up the algorithm have been investigated.

5.4.2 Speed optimizations

Being the most promising option to speed up the algorithm, a method called Marginal Cost Calculation has been adopted. This method defines how to calculate the change in total coordination costs instead of recalculating the entire total coordination cost, which should lead to speed improvements [MAC06]. Implementing this method indeed caused a significant improvement, which becomes most obvious with large input sizes: clustering 200 modules takes approximately 100 seconds. Later tests with input from JAR files of real software projects with up to 1200 modules showed that the algorithm could not have produced results in a reasonably time without this optimization.

5.4.3 Algorithm parameters

The algorithm has a few key parameters which have been or may be modified to optimize the speed or functionality of the algorithm. These are specified in this paragraph.

First, since the initial number of clusters (originally equal to the number of modules) provides an upper bound for the number of clusters after execution of the algorithm (as no new clusters are ever created), a maximum can be set for the number of clusters in the algorithm output. For testing, the algorithm has been modified to create 12 initial clusters (instead of a cluster for each module), initially assigning each module to cluster number (module number % 12). This initial clustering is, of course, quite arbitrary, and the stable limit (specified next) has to be set high enough to be sure that this randomness is gone in the algorithms output.

Second, the stable limit, the number of sequential unsuccessful attempts to decrease the total coordination cost before the
algorithm ‘gives up’, provides a balance between the speed and quality of result of the algorithm. [FER98] specifies a value of 50, but there is no reason to assume that this value is suitable for software systems. [MAC06] specifies a value equal to the size of the DSM, but this makes the algorithm far too slow. Test results show that most of the reduction in total coordination cost is already reached when using a relatively low limit; the relationship is shown in Figure 4.

![Figure 4. Influence of the Stable Limit on the Total Coordination Costs and running time. For x=0 y=initial TCC, For x->∞ TCC reaches a minimum, unreachable in practice due to processing time requirements.](image)

Because there is no suitable limit for all types of input, it is better to monitor the reduction in total cost change during execution of the algorithm, and stop if there is no longer any structural reduction. This concept has been implemented for testing purposes. Test results showed that if the algorithm stops when the total decline of total coordination cost over the last x attempts gets below 1/10000 of the initial total coordination cost, the moment the algorithm ‘stops’ is better chosen. This approach does, however, slow down the algorithm, as the sum of x numbers has to be calculated after each cycle. X needs to be set large enough to prevent incidental bad cycles to stop the algorithm; for testing x was set to a quarter of the number of software modules in the input matrix. A larger value is more reliable but also slower.

Third, the pow_cc variable that indicates the ‘penalty’ for large clusters can be changed to create a larger number of smaller clusters (or a smaller number of larger clusters). Tests have been conducted to see the effects of changing the optimal value specified in the literature, namely 2. These tests did not show that a different number works significantly better. The difference in results when using 1 or 2 does not seem to vary too much, while a higher penalty tends to create too many clusters. It is recommended to leave the value at 2.

### 6. Diagram Development

To develop a comprehensible diagram, a suitable definition of ‘comprehensible’ needs to be found. For this purpose, information on research that addresses the comprehensibility of different options for diagrams is needed. Fortunately, results of research concerning this issue have been published [MO00]. The diagram development process is based upon this research. Also, human media interaction considerations from [DIX04] have been taken into account.

#### 6.1 Diagram Goal

The goal of the diagram style developed in this research is to aid the reader of the diagram in understanding a software development process at a high level. For this purpose, the diagram style will contain constructs for software module interdependencies and software-developer dependencies. In general, target readers are those that have in some other way interest in the development process, but are not actively involved in programming activities or lack technical knowledge to understand these. This can for example be software development company managers or customers of software systems being developed.

#### 6.2 Diagram Requirements & Solutions

Several diagram style requirements and ‘good practices’ can be extracted from the available research mentioned and from the diagram goals. These are presented in this paragraph and have also been validated by discussing these and other alternatives with fellow students.

#### 6.3 Requirements Specification

A summary of requirements for different elements of the diagram style are presented in this section.

##### 6.3.1 Developer representation

The following requirements concern the representation of individual developers in the diagram style:

- To prevent confusion, developers should be clearly distinguishable from other parts of the diagram. This can be accomplished by giving them a unique color and shape or icon [DIX04]
- Concerning the limited size of software development teams, all individual developers of a team should be represented.
- Developers should be identifiable (i.e. by their initials). This is just a small amount of information per developer which will present no problems.

##### 6.3.2 Software modules

The following aspects are taken into account when representing clusters of software modules:

- Software module clusters should be easily distinguishable from other parts of the system, for the same reason as mentioned for developers.
- The size of a software cluster should be visible. The size of a cluster in the diagram can represent the actual number of modules contained in the module, or the total size (Kbytes) of the modules in a cluster.
- The content of a software cluster should be identifiable. Since putting this information in the diagram may make it too complex, cluster representations may be made clickable or may have a tooltip displaying their content. This information should also remain available when a diagram is printed.

##### 6.3.3 Dependency representation

The dependencies between different software clusters and between developers and clusters are another critical element of the diagram style begin developed. The following requirements should be met:

- The ‘strongness’ of a dependency should be visible, i.e. by line thickness or length.
- There should be as less as possible crossings of connections to reduce the risk of misinterpretation.

#### 6.4 Diagram example

Taking into account the specified requirements, an example of a diagram, representing an online chess game (where the game logic is on the server side) could look as displayed in Figure 5. Note that although the diagram is actually in color a printed version of this paper may not be.
In the diagram, five developers are presented on the inner (imaginary) circle, while seven software module clusters are presented on the outer circle. The difference in size of clusters is clearly visible. Black lines (interrupted for clarity) indicate dependencies between developer and cluster, while lines between outer circle items (supposedly red) indicate dependencies between clusters. These two types of dependencies can be extracted from the DSMS and clustering information. Lines between developers (supposedly blue) indicate dependencies between developers, which are a logical consequence of the other dependencies: the dependency of two developers can be calculated as the sum of dependencies between software clusters the two developers are working on.

Figure 5. Example Diagram of a fictional online chess game

7. DEVELOPED TOOL

As mentioned, during the research presented in this paper a tool has been developed to be able to test clustering algorithms. This chapter describes how this tool works and how its modular design makes the tool’s code reusable. Also, the results of integration test in the TESNA tool are presented.

7.1 Tool Architecture

The tools’ graphical user interface uses a Model-View-Controller architecture to distinguish between the interface Specification and technical features. [LET05]. Several important interfaces and extension mechanisms have been used to ensure modularity [NIN05]. Specifically, clustering algorithms are implemented as an implementation of the Clusterer interface, while panels for displaying diagrams in a specific style can be implemented as an implementation of the DiagramPanel interface. New implementations are linked to the application by specifying them in the Configuration interface.

7.2 TESNA Integration

Because of the fact that the results of this research are to be used in the before mentioned TESNA tool the author of this paper and Chintan Amrit, PhD student at the MB faculty of the University of Twente, have cooperated in an attempt to perform this integration. By adding a few methods for data structure conversions, the integration was relatively easy, allowing TESNA to use the MacCormackClusterer to cluster information in DSMs generated from JAR files of real open source software systems. This breakthrough is very important, as it implies that the results of this research can easily be used during a software development process, by generating a JAR file of a project and then using TESNA to generate a diagram. Initial test results showed that the algorithm parameters need to be tweaked to better work with ‘real’ input. A lot of time has been spent on optimizing the way the tool stores results, because OutOfMemoryErrors were occurring too often. Finally, we managed to get the clustering to work for JEdit [JED07], an editor consisting of 1204 modules. The tool was able to cluster the modules, display the results and export these to a GIF file in approximately 10 minutes, requiring 750 megabytes of RAM. The surprisingly huge memory consumption is mainly caused by Java’s ImageIO library that writes the PNG file out of a 1204×1204 JTable. Finally, it was possible to create diagrams starting to look similar to the example displayed in Figure 5. An example of such diagram is shown in Figure 7. Because this diagram displays data from one development day, not all clusters have developer relationships since there are clusters on which no work was done on that day. Of course, beyond the scope of this research, more programming work needs to be done before diagrams can actually be generated in the developed style. Now that the research has been done, however, this should be relatively easy to accomplish.

8. CONCLUSION

The research presented in this paper has yielded the following answer to the research question: development process of a software system can be comprehensibly represented by clustering together groups of dependent software modules from the system and displaying the relationships between clusters and between clusters and developers, as well as dependencies between developers that arise in this way, using a diagram style that takes into account scientific research on information representation.
9. DISCUSSION & FUTURE WORK

In this section a few final remarks on the research done are presented.

First, one might be skeptical about the actual use of the development diagrams. Comprehensible diagrams representing complex processes can significantly help in understanding the basics of the logistical aspects of these processes. This can, for example, lead to a more effective team and task structures in a software development company by placing people that work on dependent parts of a system together. This is also one of the main goals of the TESNA project. Other examples are also possible, like new developers entering a team can more easily integrate when they know who is doing what, or customers of a software system that know whom they must contact for feature or bug-fix requests.

Second, the developed tool uses a lot of system resources and time for generating diagrams of software systems of up to 1200 modules, which is not extremely big. It can be discussed how useful the algorithm is for even larger systems. Due to the non-exponential nature of the algorithm, however, systems fast enough to complete the process for over 2000 modules in a reasonable amount of time (less than 30 minutes) are commonly available today. The most important system requirement is the amount of RAM; a java heap size of 1GB or more is required.

Third, one might wonder how scalable the diagram style is. That it works for five developers and seven clusters does not necessarily imply that it also works for larger systems. The clustering algorithm, however, allows for an upper bound on the number of clusters, which should indeed not be set too high. There is also no need for a very large amount of clusters, since even in large software systems the modules of the system can always be grouped in just a few clusters if the clusters are more general [LET05][KAZ98]. The number of developers is, of course a fixed number. If this gets too big, for example in a globally distributed development process it may cause problems in diagram comprehensibility. It is possible, however, to apply the very same clustering algorithm to developers to reduce the number of items to be displayed. This could easily be implemented in the TESNA tool.

Fourth, a frequently asked question in reviews of this paper was what happens to the dependencies in Figure 6 that lie outside any cluster. It seems to be a common misconception to assume these dependencies are lost. Actually, these dependencies are the dependencies between clusters, and play a very important role in the diagram. Without any dependencies between clusters, a software system would consist of several fully independent modules, and the diagram would have no lines between clusters, which wouldn’t be very good for the software system.

ACKNOWLEDGEMENT

I would like to thank the following people who play a significant role in making this research possible, providing useful information and/or reviewing my work (in alphabetical order): Chintan Amrit, my supervisor in this research, Corrie Huijs, coordinator of the Bachelor Referaat course, Marieje Jochemsen, a very good friend, and Daniel Moody, coach of the Information Management track group.

REFERENCES


[GUE04] Guéhéneuc, Y.G. A reverse engineering tool for precise class diagrams Proceedings of the 2004 conference of the Centre for Advanced Studies on
Collaborative Research, Markham, Ontario, Canada, 2004, p 28-41


[LAU02] Lauesen, S. Software Requirements – Styles and Techniques, Addison-Wesley, Boston, USA, 2002


[NIN05] Nino, J., Hosch, F.A. An Introduction to Programming and Object Oriented Design using Java 1.5, 2nd edition, John Wiley & sons inc., USA, 2005


[STE93] Steffen, J.L., Eick, S.G. High Interaction Data Visualization using Seseoft to visualize program change history (abstract), Proceedings of the SIGCHI conference on Human factors in computer systems, Amsterdam, the Netherlands, 1993, p517


public class MacCormackClusterer implements Clusterer {

    public static final String ID = "MacCormackClusterer";
    private int pow_CC = 2; // The cluster size penalty, default=2
    private int stable_limit = 50; // number of unsuccessful cost reductions before ending.
    private int maxNumberOfClusters = 12;
    private long reAssigned = 0;
    public static final int THRESHOLD = 0;
    private double totalcost;
    protected double[][] costmatrix;

    public Set<Set<Integer>> clusterSoftwareDSM(double[][] input) {
        // SetInitialClusters divides the indices of the input array over maxNumberOfClusters clusters.
        List<List<Integer>> clusters = setInitialClusters(input);
        boolean stable = false;
        int attempt = 1;
        int maxfailed = 0;
        totalcost = calcInitialTotalCost(input, clusters);
        while (!stable) {
            //select random module from random cluster
            int sourceCluster = (int)(Math.random() * clusters.size());
            int sourceClusterIndex = (int)(Math.random() * clusters.get(sourceCluster).size());

            // find highest bid
            double bestBid = 0;
            for (int i = 0; i < clusters.size(); i++) {
                if (i != sourceCluster) {
                    double costReduction = calcCostReduction(input, clusters, sourceCluster, i, sourceClusterIndex);
                    if (costReduction < bestBid) {
                        bestBid = costReduction;
                        bestCluster = i;
                    }
                }
            }

            if (bestBid < THRESHOLD) {
                totalcost = totalcost + bestBid;
                clusters.get(bestCluster).add(clusters.get(sourceCluster).get(sourceClusterIndex));
                clusters.get(sourceCluster).remove(sourceClusterIndex);
                clusters = cleanup(clusters);
                attempt = 1;
            } else {
                if (attempt == stable_limit)
                    stable = true;
                else{
                    attempt++;
                    if (attempt > maxfailed)
                        maxfailed = attempt;
                }
            }
        }
        // Set final clusters
    }
}
// buildResultSet builds a set of set of Integers from the cluster list.
return buildResultSet(clusters);

protected double calcCostReduction(double[][] input, List<List<Integer>> clusters, int source, int dest, int elem) {
    double result = 0;
    int N = input.length; // the size of the DSM
    int n = clusters.get(dest).size() + 1; // dest cluster size after move
    int m = clusters.get(source).size(); // source cluster size before move
    double factor;
    // In to Out in source cluster
    factor = (0 - Math.pow(m, pow_CC) + Math.pow(N, pow_CC));
    for (int i = 0; i < m; i++) {
        result += input[clusters.get(source).get(i)][clusters.get(source).get(elem)] * factor;
        result += input[clusters.get(source).get(elem)][clusters.get(source).get(i)] * factor;
    }
    // Out to in
    factor = (0 - Math.pow(N, pow_CC) + Math.pow(n, pow_CC));
    for (int i = 0; i < n-1; i++) {
        result += input[clusters.get(source).get(elem)][clusters.get(dest).get(i)] * factor;
        result += input[clusters.get(dest).get(i)][clusters.get(source).get(elem)] * factor;
    }
    // source cluster dependencies
    factor = (0 - Math.pow(m, pow_CC) + Math.pow(m-1, pow_CC));
    for (int i = 0; i < m; i++)
        for (int j = 0; j < m; j++)
            if (clusters.get(source).get(i) != elem && clusters.get(source).get(j) != elem)
                result += input[clusters.get(source).get(i)][clusters.get(source).get(j)] * factor;
    // dest cluster dependencies
    factor = (0 - Math.pow(n-1, pow_CC) + Math.pow(n, pow_CC));
    for (int i = 0; i < n-1; i++)
        for (int j = 0; j < n-1; j++)
            result += input[clusters.get(dest).get(i)][clusters.get(dest).get(j)] * factor;

    return result;
}