F L O A T

Robert B. Lisek
Fundamental Research Lab

The aim of project

FLOAT is a multi-user hyperstructure kit - a set of tools or primitives for building, modeling, analysis, and visualization of hyperstructures.

The main problem that we're considering is a problem of ordering (sorting) and visualizing information. The tool should be sufficiently flexible and stiff at the same time. Flexible in the sense that the user must always have the possibility of :
• adding new content,
• re-organizing the information to which he has access,
• presenting the same information in various ways (the means of viewing of his choice, appropriate to the content). Stiff in the sense that the structure in which information is arranged is partially ordered.

We begin with a very simple set of rules for building blocks and their assembly-- one type of unit, one type of connection - and then assemble them into a variety of mechanisms.

The large scale architectural decisions (general)

In order to organize information we use **DAG directed acyclic graph** (directed graphs without directed cycle) in which their edges possess weight. Node (or cell) is any information unit: text, image, sound or other kind of data. Two nodes (units) are connected by directed edge.
Each DAG can be treated as a **POSET** (partially ordered set = a set with binary relation which is reflexive, antisymmetric and transitive).
Each poset can be draw as Hasse diagram (upward drawing DAG).
Every DAG has at least one topological sort (linear extension), and we can use depth-first search to find such an ordering. A **topological sort** of a directed acyclic graph is an ordering on the vertices such that all edges go from left to right (see below). Topological sorts are called **linear extensions** in the theory of posets.

Therefore,information may be organized and visualized in many different ways (the user may see information in many different ways):
1. as DAG
2. as Hasse diagram (updraw DAG - see chapter > Hasse diagrams)
3. as a set of linear extensions by using topological sort of poset (extension – see chapters > Topological sorting and Linear extensions)
(poset can be represented as the intersection of linear extensions,
najmniejszy taki zbior linear extensions ktorych intersection daje poset nazywany jest wymiarem)

Other important feature of FLOAT is that we can find shortest path in DAG (in which each edge has positive weight) between two freely chosen nodes. We've applied here the Dijkstra's algorithm. It is important that we search for the shortest path, but this one

that at the same time has the strongest connection. that's why in the algorithm we should use a complement of the weight w, that is to say (1-w).
.

What we call weights are numbers from range [0,1]. These numbers represent a degree of our believing that one document implicates another, in other words this is a degree of our believing that a document q has 'more information' than p or p is 'less defined' than q.
By selecting two nodes and starting Dikstras algorithm the user may find the shortest path which has in the same time strongest connection between nodes. So the user can check a probability that information that is contained in one of them implicates information contained in another. As a result the user can see a drawing of this path and a number from the range [0,1].

The specific description, math and algorithm background)

DAG- Directed Acyclic Graph **also known as a *partial order* or *poset*.**

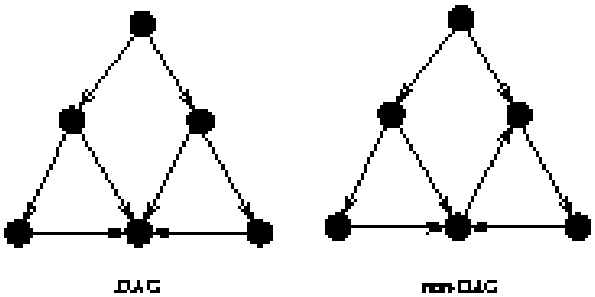


Figure: Directed acyclic and cyclic graphs

A directed, acyclic graph, or *DAG*, is a directed graph with no directed cycles. Although undirected acyclic graphs are limited to trees, DAGs can be considerably more complicated. They just have to avoid directed cycles.

A *topological sort* of a directed acyclic graph is an ordering on the vertices such that all edges go from left to right. Only an acyclic graph can have a topological sort, because a directed cycle must eventually return home to the source of the cycle. However, every DAG has at least one topological sort, and we can use depth-first search to find such an ordering. Topological sorting proves very useful in scheduling jobs in their proper sequence, as discussed in catalog Section

POSET – Partially Ordered Set

A **poset** (P,<) is set with binary relation <
which is

a<*a* (reflexivity);
if *a*<*b* and *b*<*a* then *a* = *b* (antisymmetry); and
if *a*<*b* and *b*<*c* then *a*<*c* (transitivity).

Strict and weak partial orders
In some contexts, the partial order defined above is called a **weak** (or **reflexive**) **partial order**. In these contexts a **strict** (or **irreflexive**) **partial order** is a binary relation which is (Click link for more info and facts about irreflexive) irreflexive and (Click link for more info and facts about transitive) transitive, and therefore (Click link for more info and facts about asymmetric) asymmetric. In other words, for all *a*, *b*, and *c* in *P*, we have that:
¬(*a*<*a*) (irreflexivity);
if *a*<*b* then ¬(*b*<*a*) (asymmetry); and
if *a*<*b* and *b*<*c* then *a*<*c* (transitivity).
If *R* is a weak partial order, then *R* - is the corresponding strict partial order. Similarly, every strict partial order has a corresponding weak partial order, and so the two definitions are essentially equivalent.
Strict partial orders are also useful because they correspond more directly to (Click link for more info and facts about **directed acyclic graph**) **directed acyclic graph**s (dags): every strict partial order is a dag, and the (Click link for more info and facts about transitive closure) transitive closure of a dag is both a strict partial order and also a dag itself.
IMPORTANT IS THAT WE CAN THINK ABOUT DAG AS ABOUT POSET.

Hasse Diagram of Poset

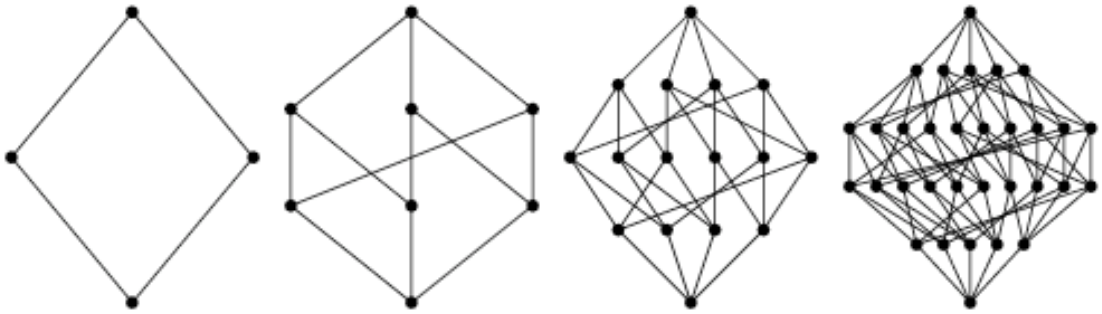
A graphical rendering of a partially ordered set displayed via the cover relation of the partially ordered set with an implied upward orientation. A point is drawn for each element of the poset, and line segments are drawn between these points according to the following two rules:

the cover relation = the transitive reflexive reduction of a partial order. An element of a poset covers another element provided that there exists no third element in the poset for which . In this case, is called an "upper cover" of and a "lower cover" of .

1. If x<y in the poset, then the point corresponding x to appears lower in the drawing than the point corresponding to y.

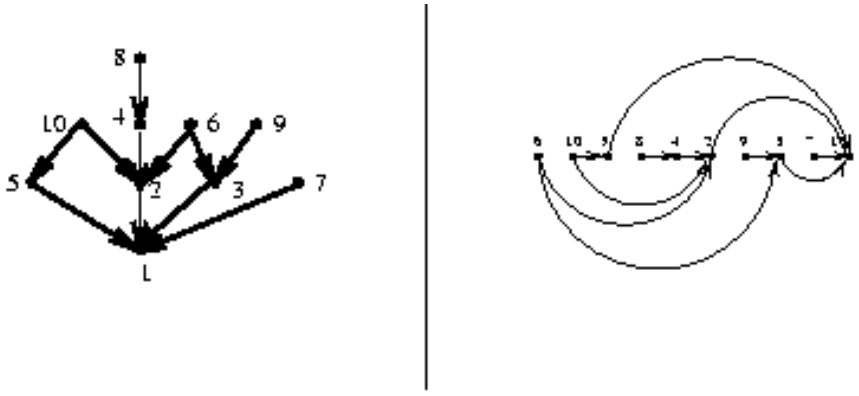
2. The line segment between the points corresponding to any two elements x and y of the poset is included in the drawing iff x covers y or y covers x.

Hasse diagrams are also called upward drawings



The above figures show the Hasse diagrams for Boolean algebras of orders , 3, 4, and 5. In particular, these figures illustrate the partition between left and right halves of the lattice, each of which is the Boolean algebra on elements

Topological Sorting



INPUT OUTPUT

Input description: A directed acyclic graph $G=(V,E)$, also known as a *partial order* or *poset*.

Problem description: Find a linear ordering of the vertices of V such that for each edge $(i,j) \in E$, vertex i is to the left of vertex j.

Discussion: Topological sorting arises as a natural subproblem in most algorithms on directed acyclic graphs. Topological sorting orders the vertices and edges of a DAG in a simple and consistent way and hence plays the same role for DAGs that depth-first search does for general graphs.

Topological sorting can be used to schedule tasks under precedence constraints. Suppose we have a set of tasks to do, but certain tasks have to be performed before other tasks. These precedence constraints form a directed acyclic graph, and any topological sort (also known as a *linear extension*) defines an order to do these tasks such that each is performed only after all of its constraints are satisfied.

Three important facts about topological sorting are:

* Only directed acyclic graphs can have linear extensions, since any directed cycle is an inherent contradiction to a linear order of tasks.
* Every DAG can be topologically sorted, so there must always be at least one schedule for any reasonable precedence constraints among jobs.
* DAGs typically allow many such schedules, especially when there are few constraints. Consider n jobs without any constraints. Any of the $n!$ permutations of the jobs constitutes a valid linear extension.

A linear extension of a given DAG is easily found in linear time. The basic algorithm performs a depth-first search of the DAG to identify the complete set of *source vertices*,

where source vertices are vertices without incoming edges. At least one such source must exist in any DAG. Note that source vertices can appear at the start of any schedule without violating any constraints. After deleting all the outgoing edges of the source vertices, we will create new source vertices, which can sit comfortably to the immediate right of the first set. We repeat until all vertices have been accounted for. Only a modest amount of care with data structures (adjacency lists and queues) is needed to make this run in $O(n+m)$ time.

This algorithm is simple enough that you should be able to code up your own implementation and expect good performance, although implementations are described below. Two special considerations are:

* *What if I need all the linear extensions, instead of just one of them?* - In certain applications, it is important to construct *all* linear extensions of a DAG. Beware, because the number of linear extensions can grow exponentially in the size of the graph. Even the problem of counting the number of linear extensions is NP-hard.

Algorithms for listing all linear extensions in a DAG are based on backtracking. They build all possible orderings from left to right, where each of the in-degree zero vertices are candidates for the next vertex. The outgoing edges from the selected vertex are deleted before moving on. Constructing truly random linear extensions is a hard problem, but pseudorandom orders can be constructed from left to right by selecting randomly among the in-degree zero vertices.

* *What if your graph is not acyclic?* - When the set of constraints is not a DAG, but it contains some inherent contradictions in the form of cycles, the natural problem becomes to find the smallest set of jobs or constraints that if eliminated leaves a DAG. These smallest sets of offending jobs (vertices) or constraints (edges) are known as the *feedback vertex set* and the *feedback arc set*, respectively, and are discussed in Section . Unfortunately, both of them are NP-complete problems.

Since the basic topological sorting algorithm will get stuck as soon as it identifies a vertex on a directed cycle, we can delete the offending edge or vertex and continue. This quick-and-dirty heuristic will eventually leave a DAG.

Implementations: Many textbooks contain implementations of topological sorting, including [MS91] (see Section) and [Sed92] (see Section). LEDA (see Section) includes a linear-time implementation of topological sorting in C++.

XTango (see Section) is an algorithm animation system for UNIX and X-windows, which includes an animation of topological sorting.

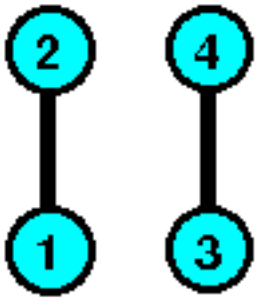
Combinatorica [Ski90] provides Mathematica implementations of topological sorting and other operations on directed acyclic graphs. See Section .

Notes: Good expositions on topological sorting include [CLR90, Man89]. Brightwell and Winkler [BW91] proved that it is #P-complete to count the number of linear extensions of a partial order. The complexity class #P includes NP, so any #P-complete problem is at least NP-hard.

Related Problems: Sorting (see page), feedback edge/vertex set (see page).

Linear Extensions

A partially ordered set **P** = (**<**,*S*) is a reflexive, transitive relation on a set *S*. Another name for partially ordered set is *poset*. In the Object Server we assume that the set *S* consists of [*n*] = {1,2,...,*n*}, the first *n* integers. A *linear extension* of **P** is a permutation p1 p2···p*n* such that *i* < *j* implies that p*i* **<** p*j*.



<2&3<4] height=134 width=119> On the left we show the Hasse diagram of a 4 elements poset. According to the definition, its linear extensions consist of those permutations of {1,2,3,4} that have 1 to the left of 2, and 3 to the left of 4. Thus, its 6 linear extensions are 1234, 1324, 1342, 3124, 3142, 3412. If Gray code output is requested, then each extension differs from its predecessor by one or two adjacent transpositions. In general, there is no listing in which successive extensions differ by a single transposition (adjacent or not).

The algorithm used to generate extensions in "lex" order is from the paper Y.Varol and D.Rotem, "An algorithm to generate all topological sorting arrangements", *The Computer Journal*, 24 (1981) 83-84. We use a recursive implementation as explained in the book F. Ruskey, *Combinatorial Generation*, manuscript, 1995. If Gray code output is requested, then the algorithm used is from the paper: G. Pruesse and F. Ruskey, "Generating Linear Extensions Fast", *SIAM J. Computing*, 23 (1994) 373-386.

Computing Prob(x<y)
It is often useful to compute Prob(*x*<*y*), the probability that *x* appears to the left of *y* in a randomly chosen linear extension. In COS we compute a table whose (*x*,*y*) entry is the number of extensions in which *x* precedes *y*; to obtain Prob(*x*<*y*) divide by the number of extensions. For the poset shown above the table is shown below.

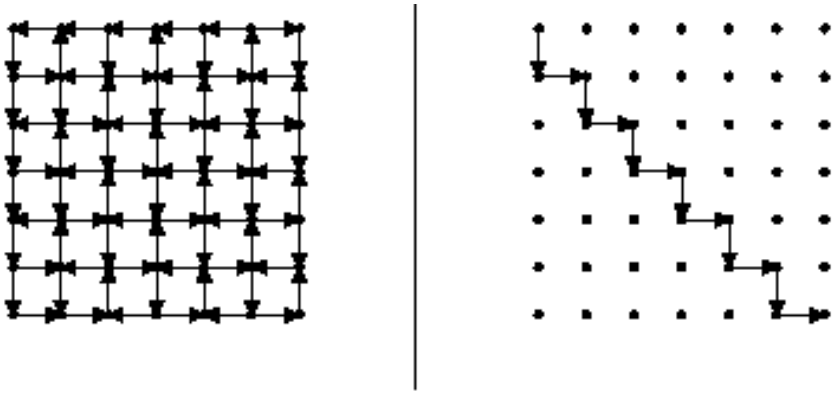
***x* =**	**1**	**2**	**3**	**4**
***y* = 1**	0	6	3	5
2	0	0	1	3
3	3	5	0	6
4	1	3	0	0

COS also outputs a pair (*x*,*y*) for which Prob(*x*<*y*) is as close to 1/2 as possible, the table entry for that pair, and the total number of extensions.

Dimension of Poset

The **dimension** of a poset *P*, denoted dim (*P*), is the least *t* so that *P* is the intersection of *t* linear orders. Examples: see Fig. 01 and Fig.02 in the chapter ‘Viewing”

Shortest Path 1



INPUT OUTPUT

Input description: An edge-weighted graph *G*, with start vertex *s* and end vertex *t*.

Problem description: Find the shortest path from *s* to *t* in *G*.

Discussion: The problem of finding shortest paths in a graph has a surprising variety of applications:

* The most obvious applications arise in transportation or communications, such as finding the best route to drive between Chicago and Phoenix or figuring how to direct packets to a destination across a network.

* Suppose we want to draw an informative picture of a graph. The center of the page should coorespond to the ``center'' of the graph, whatever that means. A good definition of the center is the vertex that minimizes the maximum distance to any other vertex in the graph. Finding this center point requires knowing the distance (i.e. shortest path) between all pairs of vertices.

The primary algorithm for finding shortest paths is *Dijkstra's algorithm*, which efficiently finds the shortest paths from a given vertex *x* to all *n-1* other vertices. Dijkstra's algorithm starts from *x*. In each iteration, it identifies a new vertex *v* for which the shortest path from *x* to *v* is known. We maintain a set of vertices *S* to which we currently know the shortest path from *v*, and this set grows by one vertex in each iteration. In each iteration, we identify the edge (*u*,*v*) where $u \in S$ and $v \in V - S$ such that

$$dist(x,u) + weight(u,v) = \min_{(u',v') \in E} dist(x,u') + weight(u',v')$$

This edge (*u*,*v*) gets added to a *shortest path tree*, whose root is *x* and which describes all the shortest paths from *x*.

The straightforward implementation of this algorithm is $O(m\ n)$. However, with simple data structures it can be reduced to $O(n^2)$ or $O(m \lg n)$ time. Theoretically faster times can be achieved using significantly more complicated data structures, as described below. If we are just interested in knowing the shortest path from *x* to *y*, simply stop as soon as *y* enters *S*.

Dijkstra's algorithm is the right choice for single-source shortest path on positively weighted graphs. However, special circumstances dictate different choices:

* *Is your graph weighted or unweighted?* - If your graph is unweighted, a simple breadth-first search starting from the source vertex will find the shortest path in linear time. It is only when edges have different weights that you need more sophisticated algorithms. Breadth-first search is both simpler and faster than Dijkstra's algorithm.
* *Does your graph have negative cost weights?* - Dijkstra's algorithm assumes that all edges have positive cost. If your graph has edges with negative weights, you must use the more general but less efficient Bellman-Ford algorithm. If your graph has a cycle of negative cost, than the shortest path between any two vertices in the graph is not defined, since we can detour to the negative cost cycle and loop around it an arbitrary number of times, making the total cost as small as desired. Note that adding the same amount of weight to each edge to make it positive and running Dijkstra's algorithm *does not* find the shortest path in the original graph, since paths that use more edges will be rejected in favor of longer paths using fewer edges.

Why might one ever need to find shortest paths in graphs with negative cost edges? An interesting application comes in currency speculation. Construct a graph where each vertex is a nation and there is an edge weighted $\lg(w(x,y))$ from *x* to *y* if the exchange rate from currency *x* to currency *y* is $w(x,y)$. In *arbitrage*, we seek a cycle to convert currencies so that we end up with more money than we started with. For example, if the exchange rates are 12 pesos per dollar, 5 pesos per franc, and 2 francs per dollar, by simply moving money around we can convert $1 to $1.20. In fact, there will be a profit opportunity whenever there exists a negative cost cycle in this weighted graph.
* *Is your input a set of geometric obstacles instead of a graph?* - If you seek the shortest path between two points in a geometric setting, like an obstacle-filled room, you may either convert your problem into a graph of distances and feed it to Dijkstra's algorithm or use a more efficient geometric algorithm to compute the shortest path directly from the arrangement of obstacles. For such geometric algorithms, see Section on motion planning.
* *Does your graph have cycles in it, or is it a DAG?* - If your graph is a directed acyclic graph (DAG), than the shortest path can be found in linear time. Perform a topological sort to order the vertices such that all edges go from left to right, then do dynamic programming on the left-to-right ordered vertices. Indeed, most dynamic programming problems can be easily formulated as shortest paths on specific DAGs. The algorithm is discussed in [Man89] if you cannot figure it out from this description. Note that the same algorithm (replacing with) also suffices for finding the *longest path* in a DAG. which is useful in many applications like scheduling (see Section).
* *Do you need the shortest path between all pairs of points?* - If you are interested in the shortest path between all pairs of vertices, one solution is to run Dijkstra *n* times, once with each vertex as the source. However, the Floyd-Warshall algorithm is a slick dynamic programming algorithm for all-pairs shortest path, which is faster and easier to program than Dijkstra and which works with negative cost edges (but not

cycles). It is discussed more thoroughly in Section . Let *M* denote the distance matrix, where $M_{ij} = \infty$ if there is no edge (*i*,*j*).

$D^0 = M$

for *k* = 1 to *n* do
for *i* = 1 to *n* do
for *j* = 1 to *n* do

$$D^k_{ij} = \min(D^{k-1}_{ij}, D^{k-1}_{ik} + D^{k-1}_{kj})$$

Return D^n

The key to understanding Floyd's algorithm is that D^k_{ij} denotes ``the length of the shortest path from *i* to *j* that goes through at most *k* intermediate vertices.'' Note that $O(n^2)$ space suffices, since we need keep only D^k and D^{k-1} around at time *k*.
*

Implementations: The highest performance code (for both Dijkstra and Bellman-Ford) available for finding shortest paths in graphs is SPLIB [CGR93], developed in C language by Cherkassky, Goldberg, and Radzik. They report solving instances with over one million vertices in under two minutes on a Sun Sparc-10 workstation. Their codes are available from http://www.neci.nj.nec.com/homepages/avg.html for noncommercial use.

LEDA (see Section) provides good implementations in C++ for all of the shortest-path algorithms we have discussed, including Dijkstra, Bellman-Ford, and Floyd's algorithms.

Pascal implementations of Dijkstra, Bellman-Ford, and Floyd's algorithms are given in [SDK83]. See Section .

XTango (see Section) includes animations of both Dijkstra's and Floyd's shortest-path algorithms.

Combinatorica [Ski90] provides Mathematica implementations of Dijkstra's and Floyd's algorithms for shortest paths, acyclicity testing, and girth computation for directed/undirected and weighted/unweighted graphs. See Section .

The Stanford GraphBase (see Section) contains an implementation of Dijkstra's algorithm, used for computing word ladders in a graph defined by five-letter words, as well as an implementation of a program to bound the girth of graphs. Algorithm 562 [Pap80] of the *Collected Algorithms of the ACM* is a Fortran program to find shortest paths in graphs (see Section).

Notes: Good expositions on Dijkstra's algorithm [Dij59] and Floyd's all-pairs-shortest-path algorithm [Flo62] include [Baa88, CLR90, Man89]. Good expositions of the

Bellman-Ford algorithm [Bel58, FF62] are slightly rarer, but include [CLR90, Eve79a, Law76]. Expositions on finding the shortest path in a DAG include [Law76].

A survey on shortest-path algorithms with 222 references appears in [DP84]. Included are citations to algorithms for related path problems, like finding the *k*th-shortest path and shortest paths when edge costs vary with time. Expositions on finding the *k*th-shortest path include [Law76].
The theoretically fastest algorithms known for single-source shortest path for positive edge weight graphs are variations of Dijkstra's algorithm with Fibonacci heaps [FT87]. Experimental studies of shortest-path algorithms include [DF79, DGKK79]. However, these experiments were done before Fibonacci heaps were developed. See [CGR93] for a more recent study.
Theoretically faster algorithms exist when the weights of the edges are small; i.e. their absolute values are each bounded by *W*. For positive edge weights, the single-source-shortest-path can be found in [AMOT88], while suffices for graphs with negative edge weights [GT89]

Shortest Path 2 appendix

Topological ordering and shortest paths, Topological ordering and acyclic graphs

---

VIEWING

User has access to different ways of viewing the graph saved in the base. Any user works with individual (subjective) vision or a fragment of the universal graph saved in the database. In this sense any user watches another fragment of the graph or another slice of the graph on his monitor. The information about graph (tops and links) is in the MSQL database.

V1. DAG
The user can see the DAG. As it may be a large structure, the number of nodes that it may present should be limited [30 nodes or less or should be set numerically (a window in which we write a number of nodes to be presented)]. In this option there is also a possibility of zooming a fragment of the graph (see fig. 05).

V2. Diagram Hassego / ordered set
An up-ordered version of V2, where we have a reorientation of the nodes arrangement (see def of Hasse diagram). All the edges are directed upwards. It means that the nodes with more general information are placed higher than their neighbours, containing more detailed information.
V3. A space-oriented viewing and interpretation (a multi-dimensional viewing)
User can see the Hasse's diagrams as the intersection of its linear extensions. So, the viewing in this mode is cutting the graphs so that we have each linear extension in new dimension.

This is very important form of presenting information in the graph, because we can see the lists that for the first view are invisible when the graph is presented like in V1-V2. Next the user can pass with the cursor through successive positions of the list (respective titles of the nodes are magnified then).

Fig. 01 The 4-element set viewed: as the digraph, as Hasse diagram, as the intersection of two linear extensions (2 dimensions), up-ordered linear extensions, right-ordered linear extensions

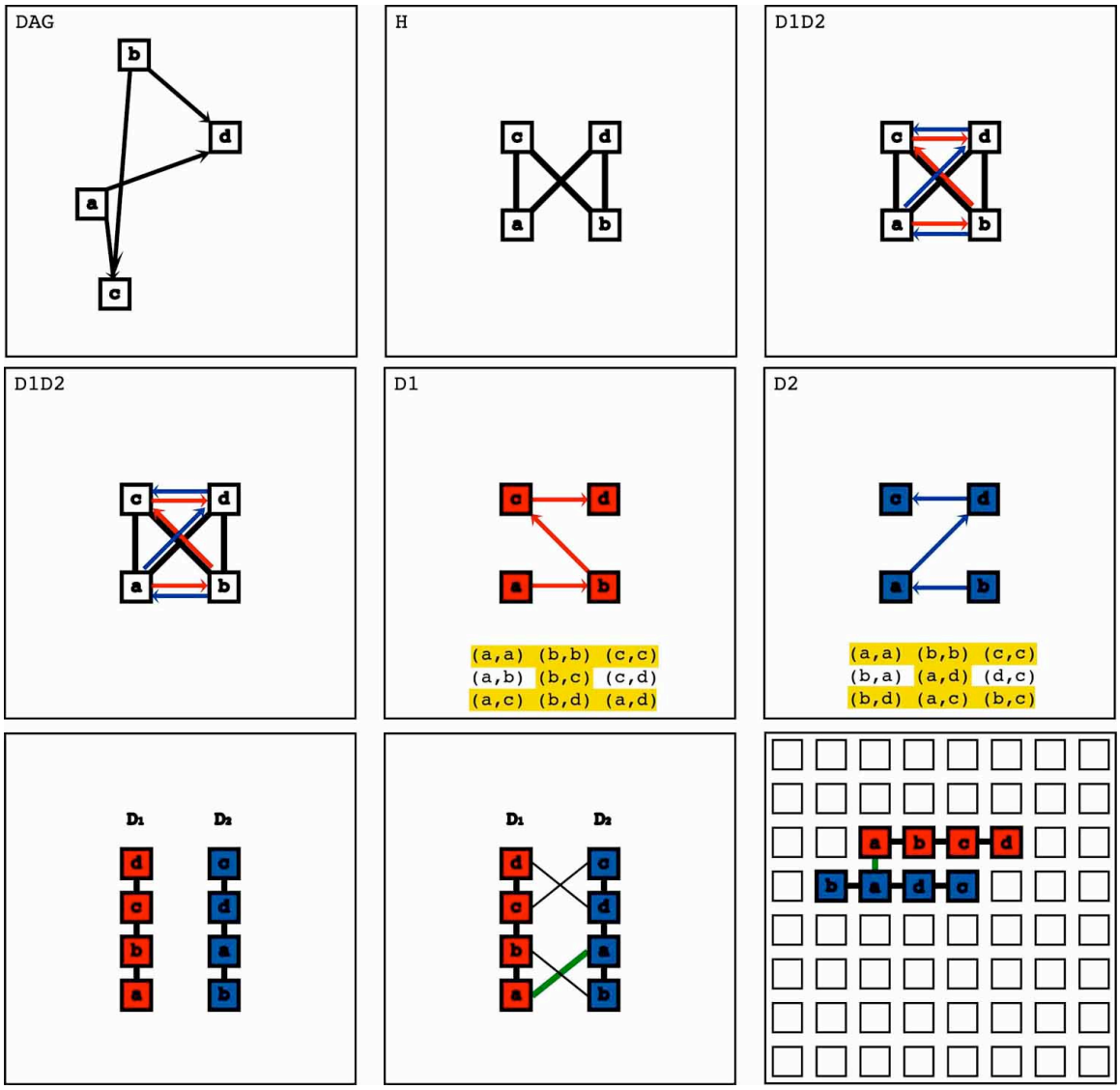
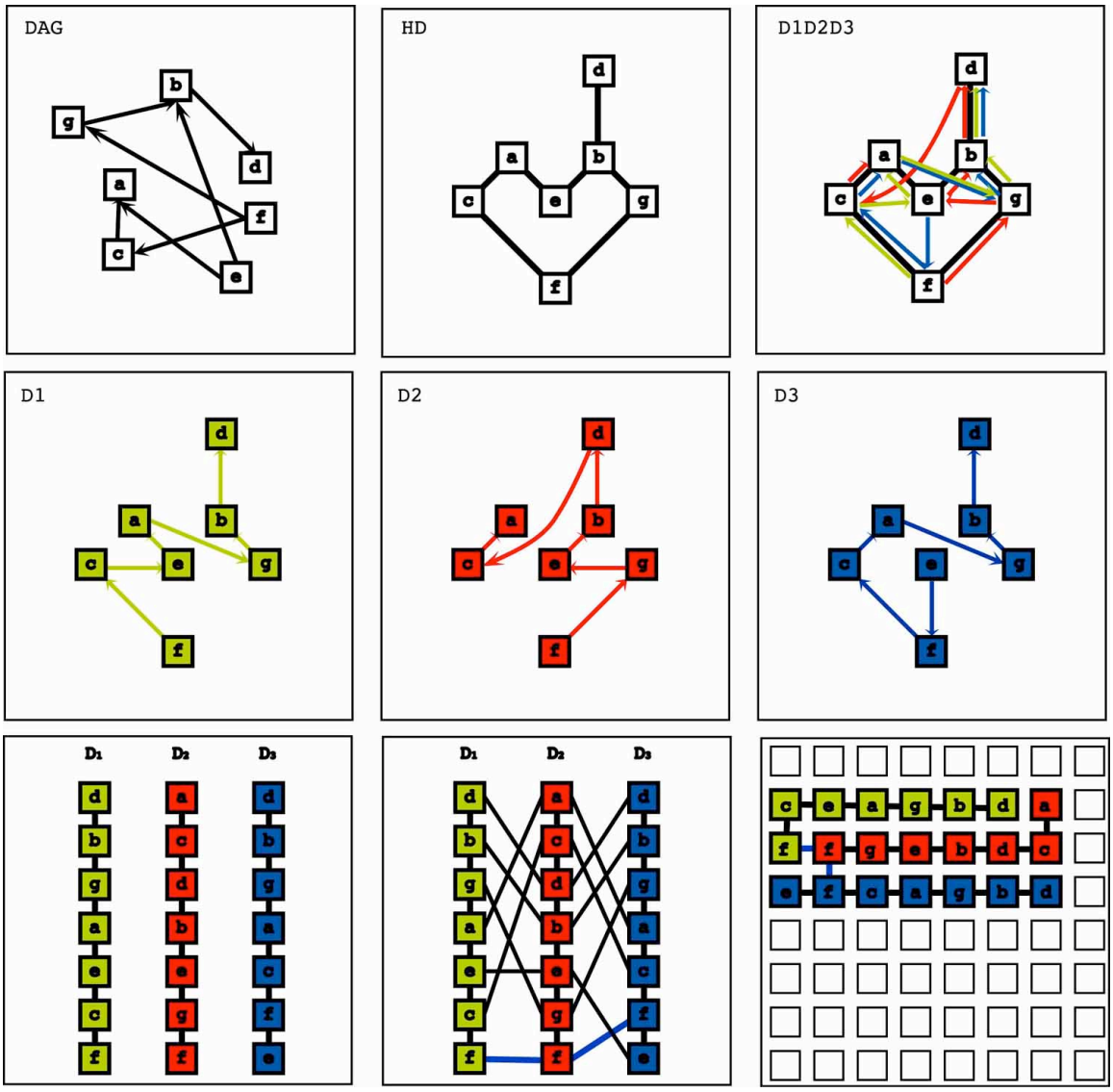


Fig 02 (below) The 7-element set as the digraph, as Hasse diagram, as the intersection of 3 linear extensions (3 dimensions): view in dimension d1,d2,d3, up-ordered linear extensions, right-ordered linear extensions



V2. Cursor-centric viewing **MOST IMPORTANT**

The most important is a way of viewing in which he sees all the neighbours of the selected node and the neighbours' neigbours (see picture). In hasse diagram the nodes containing more general information are placed above of their neighbours, that contain more detailed information.

The dynamic of the selection process and viewing information consists in existing of DYNAMIC MECHANISM OF CENTERING (FOCUSING THE ATTENTION): presents nodes connected with a selectioned node around it (above all, it presents nodes with detailed information, as far as possible it presents them below of the selectionned node). After selecting the node, the node and title is highlighted with a colour.
Next, while clicking the node is automatically centered on the screen, and the node title first is magnified, then it returns to its normal size and is highlighted by a colour (this movement should be sufficiently suggestive and look like floating on the graph

structure). The nearest neighbours are placed aside. The rest of nodes are invisible and placed on the horizon see picture).

After the selection of the node that is at the bottom of the screen it is placed on the center of the screen and nodes-neighbours that are placed below are automatically added. In this way user can travel through all the graph.

Fig 03. the movement of cursor, selected nodes is blowing up, 2 versions

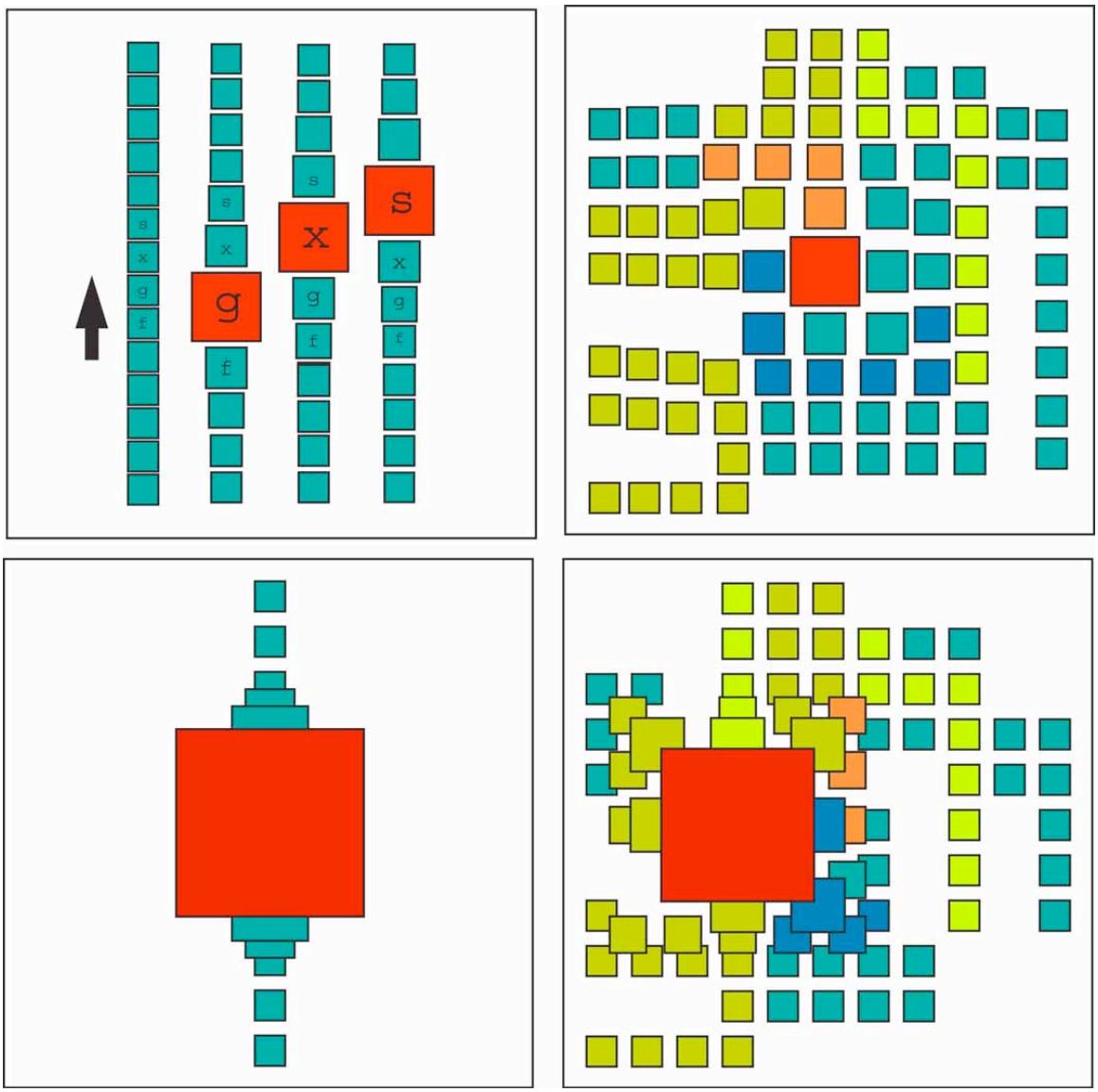


Fig.04 The example of nodes with titles.

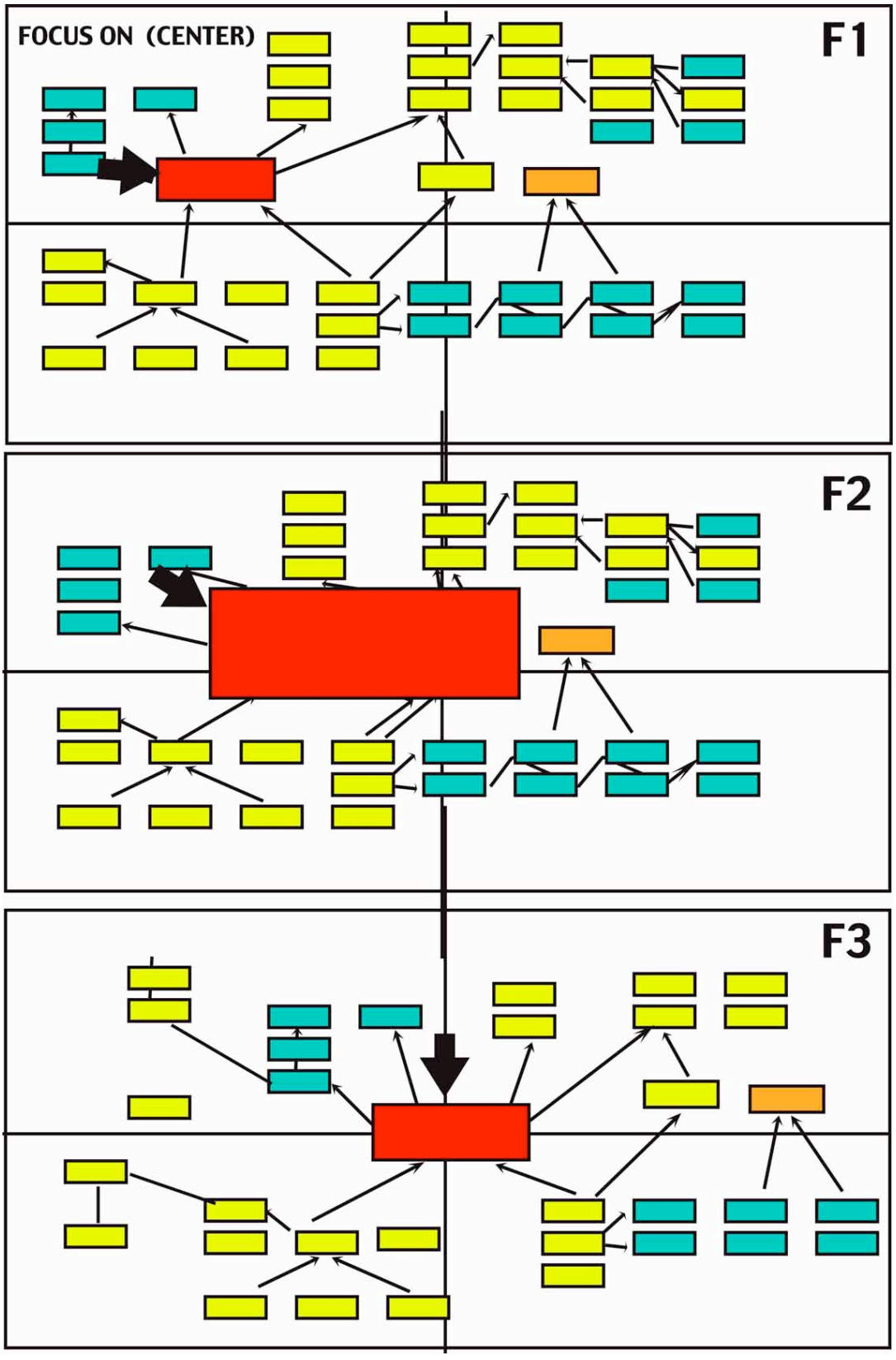


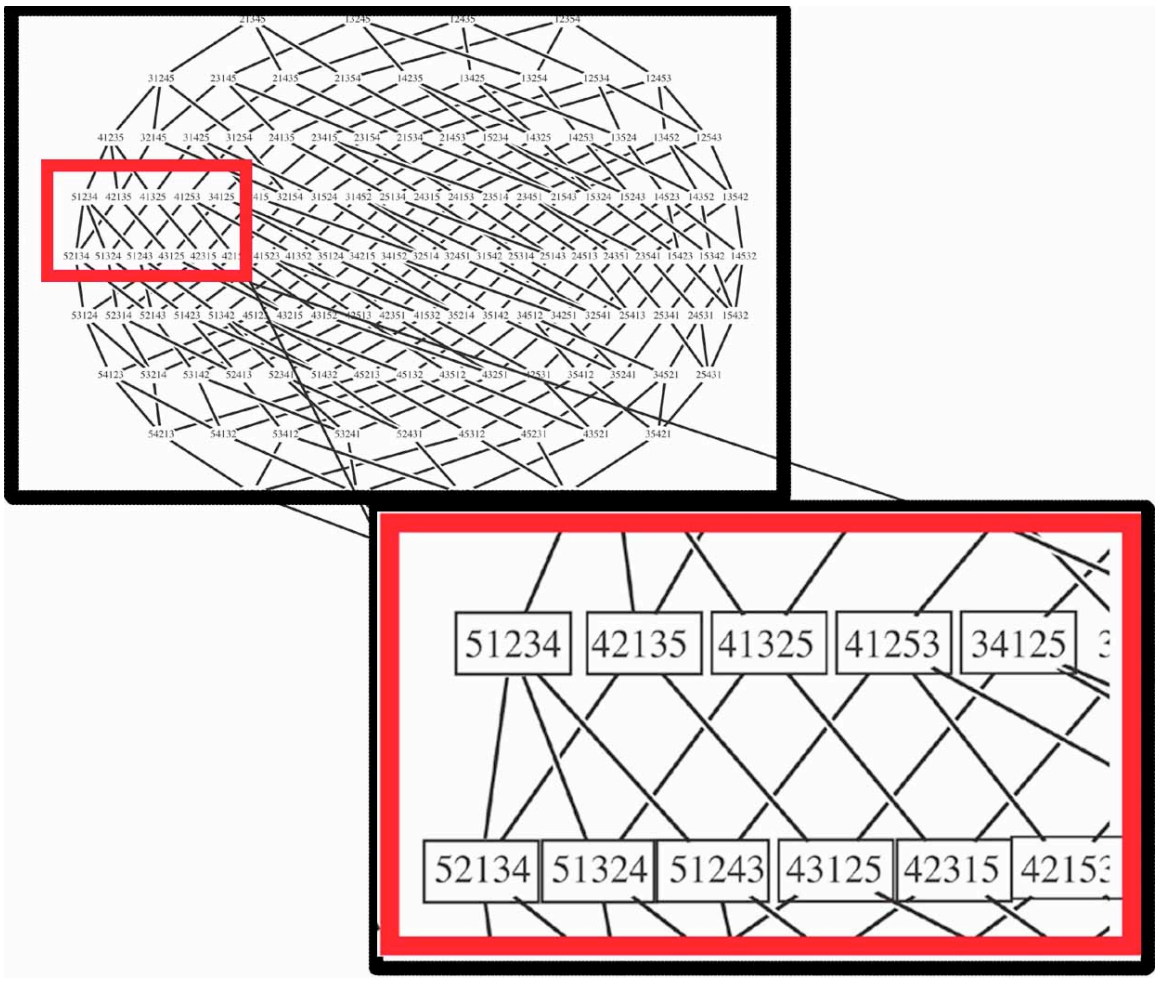
Fig. 05. Dynamic mechanism of focusing and centering

EDITION

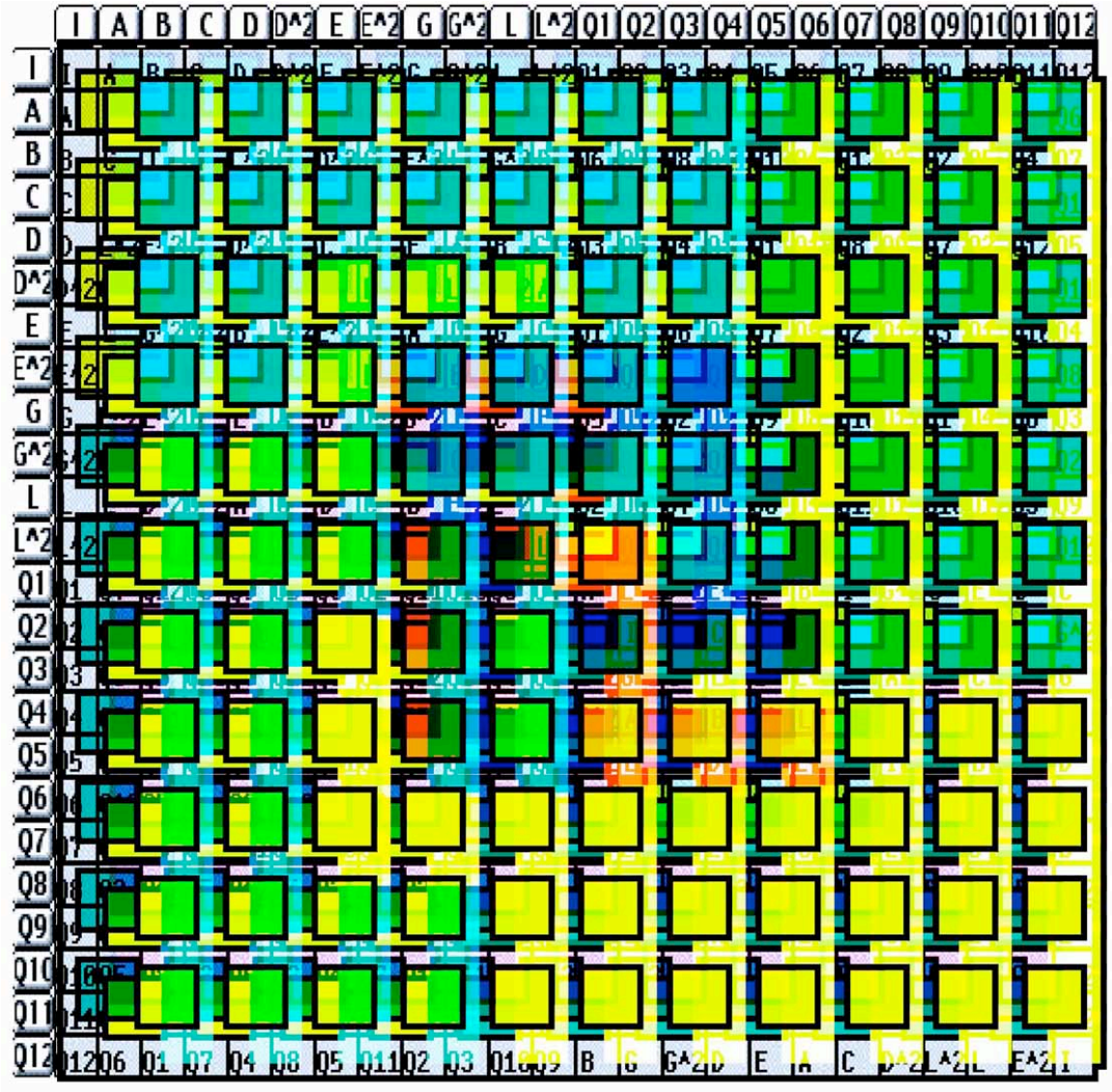
For the graph edition we need to pass to the edition mode. The user can add a node, add a link, change its weight and create a new clone of the node (create a new version of document). Restriction: a user can delete only these nodes and links that he created before himself. Information about new nodes and added links, clones of the nodes are sent to the base.
Users open html sites of the project FLOAT , with its description and indications about use.
On the next page they chose nickname and passwords and they send this information to the server. Next they go through the page where there is a link downloading FLOAT as Java web start application
After having log in, a window with a DAG graph is opened
In this window they can chose appropriate keys:
• see graph as: 1.DAG, 2.Hasse diagram, 3. multi-dimensional view : as the intersection of its linear extensions
• select node
• focusing on and center to the selectioned node
• zoom a fragment of the graph
• rotate he graph around
• add nodes
• add links
• change weight of link
• find the shortest path between two nodes
• change a content of the node

EDITION of content in the node

After doubleclicking a node user open a new window in which may edit text content of node. It will be short info (3-5 proposals description of one subject or idea).
The edition of content should be placed in a new window, the user should have access to several windows simultaneously and a possibility to change window's size (he should see a window with graph and a window of the editor at least).
In the editor user may
• load document
• save
• write new texts
• add attractors and change the power of attracting, (attractors are not necessarily, but if it's possible to obtain it out from FLEXTEXT, it could be great).
• change fonts,
• add simple graphical object,
• change colors,
• change sizes of fonts,
• numeric transformations: rotation, size etc,
• change the position of the beginning of text sequence,
• delete object,
• clear.



F06. The Hasse diagram of a poset and its detail



F07. a poset as multi-dimensional object but all dimensions (linear extension) are presented in the same time on the plane.