A-Vu: A Visualization Tool for Complex Software Systems

J. C. Smart and V. Vemuri

Lawrence Livermore National Laboratory
and
University of California, Davis

Abstract

The basic concepts underlying the A-Vu utility are presented. A-Vu is a general purpose tool for visualizing the structural aspects of complex software designs. Starting from a directed graph representation of a system or an Ada program library, A-Vu develops a variety of "nice" graphs that reveal different visual representations of a design. A-Vu determines the method of visual expression, organizes the diagram in terms of desirable perceptual properties, and establishes an aesthetic layout. Formally, given a directed graph \( G = (V, E) \), A-Vu searches for a configuration \( C \) such that a suitably defined "energy" function \( J(C) \) is minimized. The definition of \( J(C) \) includes software engineering considerations such as layering and modularity, as well as aesthetic considerations like visual appeal, cognitive correctness, and so on. This function is used in conjunction with simulated annealing and genetic algorithm techniques to obtain the "minimum energy" configurations.

1: Introduction

To develop and maintain a modern software system, a programmer must possess an adequate understanding of its design and organization [11]. Software systems containing hundreds of modules are now common. Associated with rapid program growth has been a dramatic increase in program complexity. As a result, modern software development and maintenance is a formidable challenge for even the most experienced software engineers. The ability to comprehend large systems is perhaps one of the major challenges facing the software industry for the 1990s [10]. The goal of this work is the development of the tools and techniques required to untangle the structural mazes underlying these systems and aid design comprehension.

An unorganized module interconnection diagram of modest size is shown in Figure 1. This diagram was generated after a brief examination of an Ada program library. Each box in the diagram represents a software module. A line from the bottom of one box to the top of another box represents a module dependency. A module is dependent upon another module if it references any of the other module's resources (i.e. via a with or use clause). The diagram in Figure 1 conveys very little useful information as it neither captures the designer's original organizational intentions nor does it effectively reveal any of the system's important structural properties.

Figure 1. \( G_1 \): Unordered Dependency Diagram

The central idea of the A-Vu system is to determine the placement of system modules in a multi-level planar diagram that best promotes design comprehension. This reconstruction process is subject to multiple criteria and a variety of constraints. For example, one may require that the modules lie (a) within the boundaries of a design space, (b) at discrete points in the design space, predefined by row and column positions, (c) anywhere in design space, (d) such that their iconic representations on the screen do not overlap, etc. The formulation of the constraints must take into consideration the traditional software engineering practices such as modularity, layering, information hiding, and so on. Other constraints addressing perceptual groupings and aesthetic considera-
tions may also enter the picture. Examples of some other considerations are minimization of line crossings, total space (area or volume) occupied by the diagram, evenness of occupancy of modules and their dependencies in the design space, symmetry of the diagram, etc. A-Vu allows users to assign different degrees of importance to each member of its criterion and constraint set and seeks a near-optimal solution. The remainder of the paper discusses the basic concepts and techniques used by the A-Vu system to achieve this optimization goal.

2: Graph Theoretic Foundation

The theory of graphs is a useful vehicle to study the structure of complex systems [15]. A finite graph \( G = (V, E) \) is defined as a finite set of vertices \( V = \{v_1, v_2, v_3, ..., v_n\} \) and a finite set of edges \( E = \{e_1, e_2, e_3, ..., e_k\} \), with \( E \subseteq V \times V \). Here \( n = |V| \) denotes the number of vertices and \( k = |E| \) the number of edges. If the vertex pair \((v, w)\) associated with an edge \( e \) is an ordered pair, then \( G \) is a directed graph. An attributed graph is a graph in which the vertices and edges have application-dependent attributes associated with them.

An edge in a directed graph corresponds to a dependency between two nodes. An obvious data structure that captures these dependencies is the adjacency list, a list of all successors for each vertex. That is, for each vertex \( v \), we associate a list, \( \text{Adj}(v) \) of all its successors. Using this notation and terminology, a software system, at a macro level, can be defined in terms of vertices (or nodes) representing software modules and directed edges representing module dependencies. The task of understanding the modular structure of a complex system is then equivalent to the layout and visualization of an associated directed graph.

Our goal is to perform certain operations on \( G \) and display the result. One may wish to first perform a variety of graph theoretic operations [2, 14], such as, (a) subdivide the graph to reveal the layered structure of software products, (b) subdivide the graph into its strongly connected components to reveal closely interdependent modules for possible aggregation, (c) subdivide the graph by separating it at its articulation points, (d) check the planarity of a graph and subdivide it into planar subgraphs, (e) check the isomorphism of two graphs, and so on. One may also wish to perform more pragmatic operations such as (a) identify common interconnection patterns like source-to-sink paths, (b) feedback loops, (c) closed paths or cliques, (d) hub-like connections, and so on.

In order to exploit the expressive power of a graph in its visual representation, however, it is necessary to go beyond purely graph theoretic considerations. Node type, shape, size, and placement in a diagram are often used to convey significant software design information. A software system has some degree of inherent hierarchy; that is, certain modules are expected to be at certain levels. Sometimes these hierarchical relations are best understood if they are laid out in the shape of a tree structure. At other times, the purpose is better served if the intermodule dependencies are shown with a parent module at a central hub and all children laid out evenly around the perimeter of the hub. It may be instructive to associate the physical size of the icon on the screen with some measure of the complexity of the module, or perhaps to associate a meaning to the position occupied by a module (or its icon) in the screen coordinates. Sometimes, the relationships between modules are best understood if any inherent functional or structural symmetry in the problem is revealed.

To meet these needs, we associate with each graph \( G \), a space \( S \) and call it the visualization space of \( G \). This space can be continuous, discrete, or hybrid. For concreteness, A-Vu currently assumes a discretized three-dimensional grid. Software modules occupy one or more of the grid cells. These cells correspond to the vertices \( v \) of our directed graph. For each element \( v_i \in V \) we assign a unique vector \( p_i \in S \), which denotes the position of \( v_i \) in \( S \). The set \( P = \{p_1, p_2, ..., p_n\} \) contains the position vectors for every vertex in \( V \). Alternatively, the set \( P \) describes how \( G \) is embedded in \( S \). The three-tuple \( C = (G, S, P) \) is defined as the configuration of \( G \) in \( S \). In simple terms, a configuration dictates how a particular graph is to be drawn.

3: Placement Techniques

The first step toward visualization is the placement of modules in the visualization space. Optimal module placement is tantamount to formulating a complexity measure for graph configurations. This involves the selection of a function \( J \) which, when applied to a graph configuration \( C \) of \( G \), returns a single scalar value characterizing the complexity of \( C \). The A-Vu system searches for a configuration \( C \) that results in a minimum value for \( J(C) \). Our task then can be stated as follows: Given an arbitrary graph \( G = (V, E) \), find \( C = (G, S, P) \) such that \( J(C) \) is minimized. Within the context of a particular graph \( G \) and a visualization space \( S \), this problem reduces to a search for an optimal set of vertex (module) position vectors \( P \).
In general, the solution space involves many parameters. If the number of nodes is \( n = nV \), and \( S \) is an \( N \)-dimensional space, then \( nN \) scalar values which minimize \( J \) must be found. Given a graph \( G \) with \( n \) vertices and a visualization space \( S \) with \( m \) unique locations for positioning the vertices, there are

\[
\binom{m}{n} \cdot n! = \frac{m!}{(m-n)!}
\]
different ways to select the position vectors for each vertex. Generally, \( m \) will be greater than \( n \) if \( S \) is assumed to be discrete and infinite if \( S \) is a continuous space. Consequently, there is an enormous number of choices for \( P \). It is not practical to exhaustively search the entire space in order to determine which graph configuration the user finds most useful. A better search strategy must be employed. Fortunately, several near-optimal solution strategies which yield satisfactory results have been developed in recent years.

The A-Vu system uses an iterative improvement strategy to perform module placement. The simplest form of iterative improvement starts with an arbitrary initial placement of the modules and interchanges randomly selected pairs of modules. If the interchange results in an improvement of the objective function, that interchange is accepted. The procedure is terminated when no further improvement is possible. A modification to this scheme is repeated iterative improvement where the above procedure is repeated several times with different, randomly selected, initial conditions in order to avoid the possibility of getting stuck in a local optimum. The A-Vu design incorporates two popular iterative improvement techniques to accomplish its goal: simulated annealing [1, 8, 12] and genetic algorithms [4, 13].

**Simulated Annealing**

The method of simulated annealing is inspired by the statistical mechanics of gradual cooling (annealing) in condensed matter [9]. This iterative heuristic technique seeks a near optimum (say “minimum”) solution by randomly perturbing an initial configuration (viz., a set of parameters to the objective function \( J\)) and accepting all moves that result in a reduction in the value of \( J\). New sets of generated parameters continue to be accepted as long as they result in decreasing \( J\). To prevent this process from getting “trapped” in a local minimum, a parameter set which produces a higher value of \( J\) is occasionally accepted with a probability that decreases with an increase in \( J\). In many implementations of this method, the acceptance probability is given by \( \exp(-\Delta J/T) \), where \( \Delta J \) is the increase in \( J \) and \( T \) is called the temperature, a term borrowed from statistical mechanics. Initially, the temperature of the system is set sufficiently high so that most configurations are accepted. With each iteration, the temperature of the system is reduced according to a predetermined cooling schedule. As the system gradually begins to cool, fewer and fewer high-\( J \) (or high energy, in statistical mechanics parlance) configurations are accepted. At very low \( T \), the probability of accepting a move to a much higher \( J \) configuration becomes very small. Eventually, the parameter system stabilizes and the process is terminated according to rules which define a sufficiently frozen (or optimized) condition.

**Genetic Algorithms**

Genetic algorithms (GAs) are search algorithms based on the mechanics of natural selection and natural genetics [3]. Unlike traditional search methods, GAs work with a coding of a parameter set and not the parameters themselves. GAs require the parameter set of an optimization problem to be coded as finite length (typically binary) strings. The mechanics of a typical genetic algorithm involve a series of steps: random number generation, string copying, and partial string exchanges. GAs also use an objective (or energy or cost) function to evaluate the fitness of a particular parameter set. Unlike simulated annealing, however, GAs search from a population of points and not a single point.

The basic form of a genetic algorithm consists of three main parts: 1) reproduction, 2) crossover, and 3) mutation. At the start, an initial population of strings (coded parameter sets) is generated (typically at random). Reproduction is the process of replicating each of the strings in a population. A simple reproduction scheme assigns a weight to each string according to its objective function value. Strings to be replicated are then randomly selected from the population using a weighted probability. Strings which are associated with higher objective function values are replicated with a higher probability.

After the reproduction phase, each replicated string is randomly paired with another string in the population. Each pair of strings then undergoes crossover. The crossover process involves the selection of a random substring from one of the strings. This substring is then swapped with the substring in the corresponding position in the other string. For example, given two strings \( A_1 \) and \( A_2 \) each of length \( l \), an integer position \( k \) where \( 1 \leq k \leq l-1 \) is selected at random. Two new strings are created by swapping the substrings \( A_1[k+1 .. l] \) and \( A_2[k+1 .. l] \). The new set of strings then becomes the next generation.
The reproduction-crossover cycle is repeated for a finite number of generations. The string which yielded the highest objective function value at the end of the final generation is chosen as the solution. To prevent the reproduction-crossover process from accidentally eliminating a potentially useful solution, a mutation operation is introduced. An occasional alteration is made to the value at a random string position, effectively guarding against irrecoverable loss of important material.

4: Components of the Energy Function, $J$

One of the important steps in the application of simulated annealing and genetic algorithm techniques is the definition of the energy (or cost, or objective) function. For our algorithms to perform well, we must be able to generate and evaluate parameter sets quickly. The efficiency of these schemes will be highly dependent on the energy function $J$. Consequently, our search for energy functions is limited to algorithms whose computational complexity is less than or equal to, say, $O(n^2)$. The following is an initial compilation of useful energy function components.

(a) Distance Component - $J_D$

Between every two vertices $v_i$ and $v_j$, where $v_j \in Adj(v_i)$, we can define a line segment $l_{ij}$ in the visualization space, denoting the graph edge from $v_i$ to $v_j$ as shown in figure 2.

![Figure 2. Euclidean distance between two points, $p_i$ and $p_j$.](image)

The distance $D_{ij}$ between any two vertices $v_i$ and $v_j$ (assuming a 3-D space) is defined as

$$D_{ij} = \|p_i - p_j\|$$

where, as stated before, $p_i$ and $p_j$ are the positions vectors of $v_i$ and $v_j$ respectively.

If

$$E_{ij} = \begin{cases} 
1, & \text{when } v_j \in Adj(v_i); \\
0, & \text{otherwise.}
\end{cases}$$

then a distance metric $J_D$ can be defined as

$$J_D = \sum_{ij} E_{ij} D_{ij} = \sum_{ij} E_{ij} \|p_i - p_j\|$$

where $J_D$ defines the total distance between all dependent node pairs in the graph. This is essentially a measure of the total length of all edges of the graph and captures, in some sense, the complexity of a program's representation in the visualization space. Minimization with $J_D$ is effective in collapsing a graph and finding a compact configuration.

(b) Proximity Component - $J_P$

The use of $J_D$ alone, however, does not always assure that connected nodes will tend to lie within close proximity of each other. This can be observed in the following case involving three nodes $v_i$, $v_j$, and $v_k$ as shown Figure 3.

![Figure 3. Design trade off between distance and proximity](image)

Suppose a graph configuration has sufficiently cooled (or evolved) to the point where it is highly unlikely that the positions of $v_i$ and $v_k$ will change due to their minimum energy relationship with other nodes in the graph. Minimizing $J_D$ would allow $v_j$ to be positioned at any point between $v_i$ and $v_k$ since $\|v_i - v_j\| + \|v_j - v_k\|$ is a constant. This situation can be resolved by incorporating the notion of proximity into the cost function. This new term is used to cluster as many dependent nodes as possible. Let

$$C_{ij} = \begin{cases} 
1, & \text{if } D_{ij} \leq d; \\
0, & \text{otherwise.}
\end{cases}$$

where $d$ is a constant designating the maximum allowable distance between two nodes. The proximity measure between all dependent nodes in the graph is then

$$J_P = \sum_{ij} E_{ij} C_{ij}$$
(c) Edge Crossing Component - $J_C$

As seen in Figure 1, edge crossings are a major contributor to a graph's visual complexity. Efficient algorithms now exist to determine if an arbitrary graph is planar (i.e., can be drawn in a plane without any edges overlapping) and, if so, generate a planar embedding [7]. Unfortunately, large graphs, such as those underlying software systems, are frequently not planar. Furthermore, seeking a planar organization of a graph is not necessarily useful due to the loss of spatial relationships between modules that results from the planarization process. Minimization of edge crossings, however, is still a very desirable feature, particularly when used in conjunction with other cost function components. Although the edge crossing problem is NP-hard, a near-optimal configuration can generally be found using simulated annealing and genetic algorithm methods. The number of edge crossings $J_C$ in a graph configuration can be computed as follows:

Let $v_r, v_s, v_t, v_u$ be four nodes in a graph where $v_s \in \text{Adj}(v_r)$ and $v_u \in \text{Adj}(v_t)$. We can assign to each node position vectors $p_r, p_s, p_t, p_u$ as before. The edge between $v_r$ and $v_s$ can then be defined by the equation

\[ l_{rs} = p_r + \alpha (p_s - p_r) \]

and the edge between $v_t$ and $v_u$ by the equation

\[ l_{tu} = p_t + \beta (p_u - p_t) \]

where $0 \leq \alpha \leq 1$ and $0 \leq \beta \leq 1$.

The two edges $(v_r, v_s)$ and $(v_t, v_u)$ cross if $l_{rs}$ and $l_{tu}$ intersect as shown in Figure 4. In order for $l_{rs}$ and $l_{tu}$ to intersect, there must exist a point $p_c$ which lies on both $l_{rs}$ and $l_{tu}$. The point $p_c$ must satisfy the following equation:

\[ p_r + \alpha (p_s - p_r) + p_t + \beta (p_u - p_t) = p_c \]

where $0 \leq \alpha \leq 1$ and $0 \leq \beta \leq 1$. To prevent the case where two edges such as $(v_r, v_s)$ and $(v_t, v_u)$ which share a common endpoint (node), from being counted as a crossing, we further restrict $\alpha$ and $\beta$ to $0 < \alpha < 1$ and $0 < \beta < 1$ (i.e., $\alpha, \beta \neq 0$ and $\alpha, \beta \neq 1$). Rearranging for $\alpha$ and $\beta$ yields

\[ (p_r - p_s) \alpha + (p_u - p_t) \beta = (p_r - p_t) \alpha + (p_u - p_s) \beta \]

Further discussion can be simplified if we assume a two-dimensional visualization space (i.e., $|\alpha| = 1$). (Note: the notion of a three-dimensional edge crossing is not particularly meaningful in this application.) Since the position vectors contain both $x$ and $y$ components, the above equation can be rewritten in matrix form. Let

\[ A = \begin{pmatrix} (p_r - p_s)_x & (p_u - p_t)_x \\ (p_r - p_t)_y & (p_u - p_s)_y \end{pmatrix}, \quad \text{and} \quad x = \begin{pmatrix} \alpha \\ \beta \end{pmatrix}. \]

We then have the equivalent equation

\[ A x = B \]

If $\det(A) = 0$ then the two line segments $l_{rs}$ and $l_{tu}$ cannot possibly intersect. If $\det(A) \neq 0$, then the parameters $\alpha$ and $\beta$ can be computed from

\[ x = A^{-1} B \]

Upon solving this equation, if $0 < \alpha < 1$ and $0 < \beta < 1$, then the two line segments $l_{rs}$ and $l_{tu}$ must intersect and likewise the two edges $(v_r, v_s)$ and $(v_t, v_u)$ must cross.

For every four vertices $v_r, v_s, v_t, v_u$, the parameters $\alpha$ and $\beta$ can be computed using the above procedure. Let

\[ T_{rstu} = \begin{cases} 1, & \text{if } 0 < \alpha < 1 \text{ and } 0 < \beta < 1; \\ 0, & \text{otherwise}. \end{cases} \]

and

\[ R_{rstu} = \begin{cases} T_{rstu}, & \text{if } \det(A) \neq 0; \\ 0, & \text{otherwise}. \end{cases} \]

then the total number of edge crossings in a configuration can then be computed as follows:

The terms $E_{rs}$ and $E_{tu}$ are non-zero when there exist edges $(v_r, v_s)$ and $(v_t, v_u)$, respectively. Similarly, the term $R_{rstu}$ is non-zero whenever the edges $(v_r, v_s)$ and $(v_t, v_u)$ cross.

Note that as stated, this procedure will not consider two overlapping collinear line segments as edge crossings. The definition of $R_{rstu}$ could be suitably modified to check for this condition when $\det(A) = 0$. Alternatively, an additional energy function component could be defined which specifically checks for this condition.

(d) Bisegmentation Component - $J_B$

It is often desired to divide a graph into two subgraph such that the number of connections between
the two subgraphs is minimal. This is analogous to placing the modules in two levels of a software hierarchy, partitioning modules into two separate subsystems, or assigning tasks to two processors trying to minimize interprocessor communication cost. In order to formulate the cost (or energy) function for our minimization algorithms, let us first define a binary variable $B_i$ associated with the nodes of the graph. When $B_i = 1$, the corresponding node belongs to one subgroup or level, while nodes with $B_i = -1$ belong to the other level. Now, an expression for the total number of connections between the two levels can be written as

$$N_c = \frac{1}{4} \sum_{ij} E_{ij}(B_i - B_j)^2$$

The factor $1/4$ can be explained by noting that the term on the right side makes a contribution of $4$ to the total sum whenever there is an edge between $i$ and $j$ (i.e., $E_{ij} = 1$) and the two nodes belong to different planes (i.e., $B_i = -1$). The above can be rewritten as

$$N_c = \frac{1}{2} N_0 - \sum_{ij} E_{ij} B_i B_j$$

where

$$N_0 = \sum_{ij} E_{ij}$$

is the total number of connections in the graph. Note if the graph is to be equally partitioned into two planes (assuming an even number of nodes), we can introduce a quantity

$$K_c = \sum_i B_i$$

which represents the difference in the number of nodes in the two groups. For equipartitioning of the nodes, this quantity must vanish for a valid solution. Thus, for an optimum bisection of the nodes into two groups, the quantity

$$J_B = N_c + \lambda K_c^2$$

for $\lambda \geq 0$, is minimized.

(e) **Hierarchical Component - $J_H$**

With the exception of cyclic dependencies, a hierarchical relationship can be established whenever a dependency exists between two modules. This relationship is often depicted in line graph form as an arrow from one node to the other. Implicit in this ordering is the notion that one module (or node) belongs at a higher level in the hierarchy. A visual representation of a hierarchical configuration can be generated by locating higher level modules at higher elevations in the visualization space. To establish which portion of the visualization space corresponds to higher and lower elevations, we define a hierarchical basis vector $h$. The scaler value representing the level $L$ of a node's position $p_i$ can then be computed by:

$$L(p_i) = p_i \cdot h$$

Typically $h = (0, 1, 0)$, reflecting our desire for higher level modules to appear towards the top of the screen while lower level modules appear towards the bottom. If

$$H_{ij} = \begin{cases} 
1, & \text{if } L(p_i) > L(p_j); \\
0, & \text{otherwise.}
\end{cases}$$

then the total number of hierarchical relationships that a particular configuration contains is defined as:

$$J_H = \sum_{ij} E_{ij} H_{ij}$$

(f) **Layering - $J_L$**

Systems are frequently organized using layering concepts. A strictly layered software system is one in which all modules are dependent only on other modules at the same level or on other modules at the level directly below. A layering quantity $Y_{ij}$ can be computed for every pair of nodes $v_i$ and $v_j$ in a graph as follows:

$$Y_{ij} = \begin{cases} 
1, & \text{if } 0 \leq L(p_i) - L(p_j) \leq 1; \\
0, & \text{otherwise.}
\end{cases}$$

The number of layered relationships that exist in graph configuration is then

$$J_L = \sum_{ij} E_{ij} Y_{ij}$$

(g) **Reflectivity Component - $J_R$**

Aesthetic concerns are an important factor in graph layout. One of the most obvious characteristics of “nice” graph layout involves the use of symmetry. Symmetry appears in graphs in three rudimentary forms: reflection, translation, and rotation. The most common of these forms is reflection symmetry which exists whenever a graph possesses one or more “mirror” reflection planes. The reflectivity of a graph can be determined by counting the number of edges that reflect onto another graph edge along a specified reflection plane.

Let $P_R = (P, N)$ designate the reflection plane in $S$ where $P$ is an arbitrary point in the plane and $N$ is a normal vector to that plane. Two edges $(v_i, v_j)$ and $(v_k, v_l)$ reflect onto each other if there exists a transformation $\Gamma_R$ about $P_R$ such that position vector $p_i \equiv p_k \Gamma_R$ and
The transformation \( \Gamma_R \) can be generated by translating the point \( P \) to the origin, rotating about the \( x \)-axis until \( N \) lies in the \( xz \) plane, rotating the space about the \( y \)-axis until \( N \) lies along the \( x \)-axis, performing the mirror reflection, and then performing the inverse rotations and inverse translation. These steps are described below and illustrated in Figure 6.

**Figure 5. Mirror reflection about \( P_R \).**

Using homogeneous coordinates \([5]\), the initial translation to move the point \( P \) to the origin is

\[
T = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -P_x & -P_y & -P_z & 1 \end{pmatrix}
\]

The inverse translation which will move \( P \) back to its original location after the rotations and reflection have been completed is

\[
T^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ P_x & P_y & P_z & 1 \end{pmatrix}
\]

The next step in the process is a rotation \( \theta \) along the \( x \)-axis. Let \( L_{yz} = \begin{pmatrix} P_x^2 + P_y^2 \end{pmatrix} \), then \( \theta = \sin^{-1} \frac{P_y}{L_{yz}} \). This resulting \( x \)-axis rotation transformation is defined as

\[
R_x = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & \sin \theta & 0 \\ 0 & -\sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & P_{yz} & P_{ly} & 0 \\ 0 & -P_{yz} & P_{ly} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}
\]

The inverse rotation transformation is then

\[
R_x^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & P_{yz} & P_{ly} & 0 \\ 0 & -P_{yz} & P_{ly} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}
\]

The normal \( N \) now lies in the \( xz \) plane. A rotation \( \phi \) about the \( y \)-axis must now be performed to align \( N \) with the \( x \)-axis. Let \( L = \begin{pmatrix} P_x^2 + P_y^2 + P_z^2 \end{pmatrix} \), then \( \phi = \sin^{-1} \frac{V_y}{L} \). This rotation and its inverse are defined as:

\[
R_y = \begin{pmatrix} \cos \phi & 0 & -\sin \phi & 0 \\ 0 & 1 & 0 & 0 \\ \sin \phi & 0 & \cos \phi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} L_{yz} & 0 & -P_y & 0 \\ 0 & 1 & 0 & 0 \\ P_y & 0 & L_{yz} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}
\]

\[
R_y^{-1} = \begin{pmatrix} L_{yz} & 0 & P_y & 0 \\ 0 & 1 & 0 & 0 \\ -P_y & 0 & L_{yz} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}
\]

With \( P_R \) now aligned with the \( yz \) plane, we are finally in a position to perform the mirror reflection. The reflection transformation is

\[
M_{yz} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}
\]

The final transformation \( \Gamma_R \) is then given by the product of the above transformations.

\[
\Gamma_R = TR_x R_y M_{yz} R_y^{-1} R_x^{-1} T^{-1}
\]

The reflectivity of a graph configuration can now be defined. Let

\[
R_{ik} = \begin{cases} 1, & \text{if } ||p_i - p_k|| \Gamma_R || < \delta \\ 0, & \text{otherwise} \end{cases}
\]

where \( \delta \) is an arbitrarily small distance. The total reflectivity of a configuration is then

\[
J_R = \frac{1}{2} \sum_{ij} \sum_{k} E_{ij} E_{ik} R_{ik} R_{jl}
\]

The above computation can often be simplified as we are frequently interested only in the reflectivity about a plane parallel to the \( yz \) plane. If the dimensions of \( S \) were \( w_x \times h_x \times d_z \), a suitable choice for \( P_R = (P, N) \) would be \((w_x/2, 0, 0), (1, 0, 0)\) as shown in Figure 5. Since \( N \) already lies along the \( x \)-axis, \( \theta = \phi = 0 \), \( \Gamma_R \) can be reduced to \( TM_{yz} T^{-1} \) or

\[
\Gamma_R = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ w_x & 0 & 0 & 1 \end{pmatrix}
\]

The transformation \( \Gamma_R \) thus maps a point \((x, y, z)\) onto \((w_x-x, y, z)\).
(h) Hybrid - J

In actual practice, it is advantageous to use more than one cost function component. The cost function used in A-Vu is the sum of several components that are selected by the user. Since some of these components may be considered more important to the user than others, a weight can be associated with each individual component. Note that some components (e.g. }J_D} and }J_L}\) operate as complexity metrics and should be minimized while other components (e.g. }J_H} and }J_R}\) act as simplicity metrics and should be maximized. Both component types are easily accommodating by assigning negative weights to the simplicity components. The resulting hybrid cost function is then defined as:

\[
J = \sum_i w_i J_i
\]

5: Illustrative Examples

This section illustrates the notion of a graph configuration and demonstrates how several of the above cost components can be applied to evaluate a graph configuration.

Example 1. Let }S_d = \mathbb{R}^2\), the set of two-dimensional integer vectors. That is, modules can only occupy positions in a 2-D space whose coordinates are integers. We wish to represent the graph }G = (V, E)\), where }V = \{v_1, v_2, v_3, v_4\}\) and }E = \{e_1, e_2, e_3, e_4\}\). The adjacency information is given either by the adjacency list as shown in the figure below or by a list of directed edges: }e_1 = (v_1, v_2), e_2 = (v_1, v_3), e_3 = (v_2, v_4), e_4 = (v_3, v_4)\) and }e_5 = (v_3, v_1)\). The configuration }C_1\) for }G\) is defined as }G, S_d, P_1\) with }P_1 = \{p_1, p_2, p_3, p_4\}\) and }p_1 = (1, 2), p_2 = (2, 2), p_3 = (1, 2), and }p_4 = (2, 1)\) as shown in Figure 6.

Suppose we are interested only in the distance, hierarchy, and reflectivity aspects of this configuration. By examining Figure 6, we can see that }J_D = 4 + 2^2\). Setting }h = (0, 1, 0)\), we have }J_H = 2. If the dimensions of }S\) are }w_D = 3 x h, \geq 2\), then the reflectivity }J_R\) about }P_R = (0, 0)\) is 1 (i.e. }e_2\) reflects onto }e_3\). By setting }w_D = 1 and }w_H = w_R = -1\), the weights for }J_D, J_H, and }J_R\) respectively, and all other weights equal to zero, the resulting value for }J\) is }1 + 2^2\).

Example 2. Suppose the visualization space is instead selected from }S_c = \mathbb{R}^2\), the set of two-dimensional real vectors. That is, the modules can now be placed anywhere in the 2-D space. With this freedom, we can define a configuration }C_2 = (G, S, P_2)\) where }P_2\) is once again a set of four vectors, as in example 1, except the }p_i\) vectors are now defined by

\[
p_1 = (1, 1),
p_2 = (2, 1),
p_3 = (1, 2),
p_4 = (\frac{3}{2}, 1 + \frac{1}{2\sqrt{2}}).
\]

Configuration }C_2 = (G, S, P_2)\) is shown in Figure 7. If we are now only interested in, say, the proximity, layering, and reflectivity aspects of the configuration, we set }w_p = w_l = w_R = -1\) and all other weights equal to zero. By examining Figure 7, we see that }J_D = 5, J_L = 2, and }J_R = 2\) (assuming }P_R\) from Example 1 and the proximity distance constant }d = 1). The resulting value for }J\) is then }-9\). Note that if we use these same weights and re-evaluate configuration }C_1\) in Figure 6, we again obtain a }J\) value of }-9\) (\)J_D = 4, J_L = 4 and }J_R = 1\). Although configuration }C_1\) exhibits a higher degree of layering than configuration }C_2\), it is not as aesthetically appealing as configuration }C_2\) due to }C_2's\) higher degree of reflectivity. A user may therefore wish to reconsider the weight set that was selected and perhaps weight }J_R\) more heavily than }J_L\). Conversely, if revealing layer structure is of greater concern, }J_R\) could be assigned a lesser weight than }J_L\), insuring a higher probability of finding configurations more closely resembling }C_1\).
A-Vu currently accepts input directly from an Ada source code program library or via its system definition language (SDL). When working with an Ada program library, A-Vu parses the program library file and extracts all the necessary module and module dependency information. This information is stored in an internal representation, based on its SDL model. Using this information, A-Vu can manipulate and display the structure of the system.

When a system description is initially loaded, a default configuration is generated using a single plane visualization space of predefined width. Each element (node) of the system (graph) is positioned in this space, filling in rows starting from left to right and top to bottom. The height of the visualization space is expanded as necessary. A-Vu then displays this default configuration, connecting each of the system elements with a line segment according to the dependency information. Figure 1 was generated in this manner.

A-Vu provides a variety of graph manipulation algorithms. System configurations can be restructured using depth-first, breadth-first searches and dependent, hierarchical, and layered arrangement algorithms. These algorithms can be applied to all, or a selected subset, of the elements in the system. A-Vu automatically displays each new configuration upon completion. Algorithms are available to find root elements (nodes with no parents), leaf elements (nodes with no dependents and only one parent), and cyclically dependent elements (elements that depend upon each other, either directly or indirectly). Some simple element pattern matching tools are provided which enable particular groups of elements to be identified and collectively repositioned. Simple graph centering and spacing algorithms are also provided.

To help analyze the complexity of each configuration, A-Vu supports a variety of metrics. A volumetric complexity window is displayed, which describes the size and dimensions of the visualization space. A connectivity metric window displays measures of total graph distance, proximity, edge crossings, hierarchy, and layering for the current configuration. These metrics correspond to \( J_D \), \( J_P \), \( J_C \), \( J_H \), and \( J_L \), respectively as defined in Section 4. A symmetry metric corresponding to \( J_R \) is also available, which describes the degree of reflectivity that the current configuration possesses. A graph metric window is provided, which displays the number of nodes, the number of edges, and the minimum and maximum degree, in degree, and out degree of the current configuration. A simple heuristic planarity measure is also displayed.

By combining these tools with the simulated annealing and genetic algorithm techniques described earlier, A-Vu searches for optimal configurations using its current suite of complexity and simplicity metrics (i.e. the various components of the energy function \( J \)). The user can construct a hybrid energy function based on a weighted sum of each of the individual components.

Figures 8 shows A-Vu at an intermediate stage of processing the system \( G_1 \) shown in Figure 1. Figure 8 was generated by A-Vu after it repositioned all operating system interface components, predefined packages, and common software tools to a different plane in the visualization space, significantly reducing the size of the optimization problem. A-Vu next found a root node and ordered the remaining components using breadth-first search. Confining each module to its existing row, A-Vu then applied the simulated annealing algorithm with a very small iteration count (i.e., \( m=2 \)), yielding Figure 10 after 34 passes. Figure 9 was displayed using A-Vu's filtering option for intra-layer element dependencies. This final figure very closely captures the structural relationships intended by the original designer. Figure 10 below shows many of the various display components of the A-Vu system in operation.
Future Research Directions

A-Vu appears to produce satisfactory results as seen by comparing Figure 1 with Figure 9. As a prototype tool, however, there are still many open issues that must be resolved.

What size and type of graph visualization space is needed to represent an arbitrary software system? What is the best method for generating new configurations? How can cooling schedule parameters be automatically selected? What are adequate tests for freeze conditions? How is the initial population determined? What is its size? How is a parameter set coded? How many generations should the GA analyze? What are suitable population conditions for terminating the algorithm? What is an appropriate mutation probability? These issues are currently under investigation.

The element most critical to our visualization process is the energy function, $J$. Additional components are being incorporated that reflect modern software engineering design criteria such as coupling, cohesion, information hiding, and other modularity constraints. Distances of nodes from the boundaries of the design space, distances between nodes and edges, symmetry of the resulting figure, design space exceedance penalty, line crossing penalty, module overlap penalty, and so on, are being considered. Other problem dependent factors such as the size of design space, designated home planes for certain modules, permitted move distances, etc. should be available as options to the user.

![Figure 10. A-Vu Components](image-url)
Acknowledgement: The authors would like to thank the Engineering Services Division and the Safeguards and Security Engineering and Computations Division at Lawrence Livermore National Laboratory for their support of this work.

References