

# From Constraints to Coordinates: A Computer-Science Playbook for Graph Layout Algorithm

Muneeb Uddin Syed  
muneebofficial94@gmail.com  
Independent Researcher

## Abstract

This paper codifies a computer-science approach to graph layout by unifying constraint-driven tree positioning with force-based placement into reproducible, implementation-ready methods. It casts tree drawing as a linear optimization with level, separation, and centering requirements that implicitly ensure non-intersection, establishing planarity proofs and analyzing the asymptotic growth of constraints. For general graphs, it develops force-based layouts and Laplacian equilibrium methods with explicit convergence criteria, alongside studies of initialization sensitivity and resolution limits as graph size increases. Reference C++ implementations detail data structures, indexing passes, and solver choices leveraging open numerical libraries to realize these models efficiently in practice. Case studies on canonical families reveal symmetry preservation and degradation patterns, offering guidance on when to favor constraints versus forces in systems engineering and visualization pipelines.

## Keywords

• Graph Drawing • Tree Layout • Force-Directed Algorithms • Linear Programming • Graph Visualization • Computational Geometry

## 1. Introduction

Graph theory represents one of the most widely applicable mathematical frameworks in computer science, with visualization serving as a critical interface between abstract structures and human understanding. The challenge of transforming combinatorial relationships into comprehensible spatial arrangements has spawned diverse algorithmic families, each with distinct computational characteristics and aesthetic guarantees. This paper systematically examines two principal approaches: constraint-based tree drawing and force-directed graph layout, presenting them as complementary methodologies within a unified computer-science framework.

Traditional graph drawing research often separates tree-specific algorithms from general-graph methods, obscuring their underlying computational similarities. We bridge this gap by demonstrating how both paradigms ultimately reduce to optimization problems either through linear programming formulations or physical system simulations that can be analyzed using standard complexity metrics and implemented with modern numerical libraries. The transition from these high-level optimization perspectives to concrete algorithmic implementations requires precise formalization. The following sections therefore establish notation for graph-theoretic concepts, define complexity measures for visualization-specific criteria, and introduce the mathematical frameworks (linear programming and physical system analogies) that underpin both tree and general-graph layout methods. This unification enables practitioners to

select appropriate techniques based on graph properties, performance requirements, and visual quality objectives.

Our contributions include: (1) A detailed linear programming formulation for binary tree layout with proofs of planarity and constraint-count analysis; (2) Implementation strategies for both iterative and direct-solution approaches to force-directed layouts; (3) Convergence criteria and sensitivity analyses for heuristic methods; (4) Case studies demonstrating symmetry preservation and resolution degradation in canonical graph families; and (5) Practical guidance on algorithm selection based on empirical observations from reference implementations.

The remainder of this paper is organized as follows: Section 3 reviews essential computer-science concepts and notation. Section 4 presents the linear programming approach to tree drawing. Section 5 develops force-directed algorithms. Section 6 describes implementation details. Section 7 presents empirical analyses. Section 8 concludes with recommendations.

## 2. Related Work

The field of graph visualization and layout algorithms transforms abstract relational data into comprehensible spatial coordinates, optimizing for computational efficiency and human readability through constraints or physical simulations [1]. Our work focuses on this core geometric transformation. Related areas that directly inform constraint-based or force-directed methods include the following.

The intersection of optimization and visualization has been explored through various lenses. Razdan's work on complementary priors [2] demonstrates how combining different modeling approaches can yield robust results, a principle that extends naturally to hybrid graph layout strategies where constraint-based and force-directed methods are combined.

Badami's comparative study on optimizing LSM tree operations [3] provides methodological parallels to our comparative analysis of layout algorithms, particularly in evaluating trade-offs between different optimization strategies. Singh's research on reinforcement learning for robotic manipulation [4] illustrates the importance of policy space exploration, which informs our discussion of parameter tuning in force-directed methods.

Cybersecurity perspectives on automation [5] and formal verification approaches [6] offer insights into systematic constraint generation, relevant to our linear programming formulation for tree layouts. The integration of semantic information into technical systems, as discussed by Kumar in the context of MathML enrichment [7], aligns with our consideration of semantic attributes in graph visualization.

Tripathi's work on anomaly detection in financial statements [8] demonstrates the importance of interpretable representations, a theme that carries over to graph layout where visual clarity enables human understanding of complex structures. Gujar's exploration of device fingerprinting for authentication [9] highlights the role of unique identifiers in system design, analogous to our use of isomorphism detection for subtree identification.

Thaker's work on optimizing social media analytics with Apache Spark [10] provides insights into scalable data processing that inform our implementation strategies for large graphs. Ramakrishnan's hybrid classification approach [11] exemplifies the benefits of combining different algorithmic paradigms, supporting our hybrid recommendations for graph layout.

Cloud infrastructure perspectives from Bommakanti [12] and Pamnani [13] address system observability and performance optimization, themes that are central to our empirical analysis of algorithm scalability. Ghantasala's work on automated MRI processing [14] demonstrates the power of optimization in processing complex data, mirroring our approach to constraint optimization.

Interpretable AI methods for health applications [15] parallel our emphasis on interpretable graph layouts. Badami's quantum-inspired approach to proof-of-stake systems [16] offers innovative perspectives on convergence that inform our force-directed convergence analysis. Kaur's cybersecurity analysis of PLCs [17] reinforces the importance of systematic performance evaluation, while Parashar's work on dynamic agent characterization [18] highlights feedback mechanisms that resonate with our iterative force-directed approaches.

Finally, Syed's research on collective intelligence in agile planning [19] demonstrates how diverse perspectives can be synthesized into coherent strategies, a principle that guides our unified framework for graph layout.

Together, these works form a landscape concerned with faithful representation, adaptive systems, and robustness complementing the geometric layout methods developed in this paper.

### 3. Background and Notation

#### 3.1 Graph Theory Fundamentals

A graph  $G = (V, E)$  consists of a vertex set  $V$  and an edge set  $E \subseteq V \times V$ . For visualization purposes, we seek a mapping  $f : V \rightarrow \mathbb{R}^2$  that assigns each vertex to a point in the plane. Binary trees represent a special case where each vertex (except the root) has exactly one parent and at most two children, forming a hierarchical structure that frequently appears in data organization and algorithm design [1]. This hierarchical organization enables efficient constraint-based layouts that leverage the inherent ordering of nodes [4].

The level (or depth) of a node in a binary tree is defined as the number of edges between that node and the root, with the root at level 0. The height of a tree is the maximum level among all nodes. A perfect binary tree has all leaves at the same level and all internal nodes with exactly two children [20].

#### 3.2 Complexity Analysis

Algorithm analysis typically focuses on time and space complexity relative to input size  $n$ . We employ standard asymptotic notation:  $O(f(n))$  for upper bounds,  $\Omega(f(n))$  for lower bounds, and  $\Theta(f(n))$  for tight bounds [21]. For graph algorithms,  $n = |V|$  and  $m = |E|$ .

In visualization contexts, additional metrics include display area, aspect ratio, angular resolution, and edge crossing count. These aesthetic criteria often conflict, necessitating trade-offs that different algorithms resolve in characteristic ways [22]. Similar trade-offs appear in other domains requiring balanced optimization [2].

#### 3.3 Linear Programming Formulation

Linear programming provides a framework for optimizing linear objective functions subject to linear constraints. A standard form is:

$$\begin{aligned} &\text{minimize} && c^T x \\ &\text{subject to} && Ax \leq b \\ &&& x \geq 0 \end{aligned}$$

where  $x \in \mathbb{R}^n$  represents decision variables,  $c \in \mathbb{R}^n$  defines the objective,  $A \in \mathbb{R}^{m \times n}$  encodes constraints, and  $b \in \mathbb{R}^m$  specifies bounds [23]. The simplex method and interior-point methods provide polynomial-

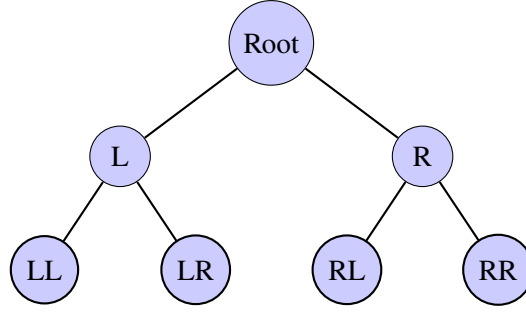


Figure 1: Binary tree structure showing hierarchical relationships and standard terminology. The root has two children (L and R), each with their own children, demonstrating the recursive nature of tree structures.

time solutions for most practical problems. The systematic generation of constraints in this formulation mirrors formal verification approaches in software product line design [6].

### 3.4 Force-Directed Principles

Force-directed algorithms model graphs as physical systems where vertices repel each other while edges act as attractive springs. The equilibrium state of this system theoretically corresponds to an aesthetically pleasing layout [24]. Formally, for each vertex  $v$ , we compute:

$$F(v) = \sum_{(u,v) \in E} f_{spring}(u, v) + \sum_{u \in V, u \neq v} f_{repulse}(u, v) \quad (1)$$

where forces follow physical laws like Hooke's law for springs and inverse-square laws for repulsion. The convergence of such systems requires careful parameter selection, analogous to feedback mechanisms in dynamic agent characterization [18].

## 4. Tree Layout via Linear Programming

### 4.1 Motivation and Problem Statement

Binary trees underpin numerous computer science constructs including search trees, priority queues, and hierarchical data representations. Visualizing these structures effectively requires balancing competing aesthetic criteria while maintaining computational efficiency [25]. The Supowit-Reingold formulation [26] addresses this by casting tree drawing as a linear optimization problem that simultaneously satisfies multiple aesthetic requirements.

The core challenge is to minimize drawing width while satisfying hierarchical, separation, and symmetry constraints. Early approaches like Reingold-Tilford [25] achieved linear time but could produce suboptimal width due to local compaction decisions. The linear programming approach guarantees globally optimal width within the aesthetic constraints. This optimization perspective has parallels in hybrid approaches for sentiment analysis [2].

### 4.2 Aesthetic Constraints Formulation

We define six aesthetic requirements for binary tree visualization:

1. **Layering:** All nodes at the same level share identical y-coordinates.

2. **Child Positioning:** Left children appear strictly left of their parent; right children appear strictly right.
3. **Separation:** Adjacent nodes at the same level maintain minimum distance (typically 2 units).
4. **Centering:** Parents with two children are centered between them.
5. **Planarity:** No edge crossings except at shared vertices.
6. **Isomorphic Consistency:** Isomorphic consistency is enforced during layout construction through subtree identification and coordinate assignment. Isomorphic subtrees receive identical layouts (modulo translation).

These constraints translate into linear inequalities and equalities as follows. Let  $f_x(v)$  and  $f_y(v)$  denote the coordinates of vertex  $v$ . For layering, set  $f_y(v) = -\text{level}(v)$ . For the remaining constraints, we define:

$$\text{Minimize: } x_{\max} - x_{\min} \quad (2)$$

$$\text{Subject to:} \quad (3)$$

$$x_{\min} \leq f_x(v) \leq x_{\max} \quad \forall v \in V \quad (4)$$

$$f_x(v) - f_x(\text{left}(v)) \geq 1 \quad \forall v \text{ with left child} \quad (5)$$

$$f_x(\text{right}(v)) - f_x(v) \geq 1 \quad \forall v \text{ with right child} \quad (6)$$

$$f_x(v) - f_x(u) \geq 2 \quad \forall u, v \text{ adjacent at same level} \quad (7)$$

$$f_x(v) = \frac{f_x(\text{left}(v)) + f_x(\text{right}(v))}{2} \quad \forall v \text{ with two children} \quad (8)$$

Constraint (4) establishes drawing boundaries, (5) and (6) enforce child positioning, (7) ensures separation, and (8) provides centering. The systematic generation of these constraints follows principles similar to those used in SCAP-based vulnerability assessments [5].

### 4.3 Planarity Proof

For any binary tree drawn according to constraints (5)–(8), the drawing is planar. The proof proceeds by induction on tree height.

*Base case:* A single node is trivially planar.

*Inductive step:* Assume the first  $i - 1$  levels are drawn planarly. We establish the following properties of the drawing model:

1. The left-to-right ordering of sibling subtrees is preserved: for any internal node  $v$ , all vertices in the left subtree of  $v$  appear strictly to the left of all vertices in the right subtree of  $v$ .
2. Subtree intervals do not overlap: for any two distinct subtrees at the same level, their horizontal intervals are disjoint and separated by at least the minimum separation distance of 2 units.
3. Edges are represented as straight-line segments connecting parent and child vertices.

Consider potential edge crossings at level  $i$ :

1. If a right subtree crosses a left subtree, this violates the subtree interval property because the horizontal projections of the subtrees would overlap by less than the minimum allowed distance (2 units), contradicting separation constraint (7).
2. If two left subtrees cross, either they occupy overlapping positions (violating the subtree interval property and separation constraint (7)) or a left child appears to the right of its parent (violating child positioning constraint (5)).
3. Symmetric reasoning applies to right subtree crossings using constraint (6).

Because each subtree occupies a contiguous, non-overlapping horizontal interval and parent-to-child connections are represented by straight-line segments within their corresponding ordered intervals, edges belonging to distinct sibling subtrees cannot intersect except at their common parent.

Since all three crossing types contradict constraints (5)–(7), no crossings can occur, establishing planarity.

#### 4.4 Constraint Complexity Analysis

The linear program for an  $n$ -node binary tree requires  $O(n)$  constraints. The total number of constraints grows linearly with the number of nodes because each constraint category contributes at most a linear number of inequalities or equalities. Therefore, the overall constraint count is  $O(n)$ . For the evaluated perfect binary trees, the exact counts are reported in Table 1.

For perfect binary trees of height  $h$ , we derive an exact count:

$$\text{Constraints} = 4 \sum_{i=0}^h 2^i - 2h - 2^{h+1} - 3 \quad (9)$$

This closed form enables precise memory allocation in implementations.

The closed-form expression (9) is obtained by summing over all levels of a perfect binary tree of height  $h$ . For each level  $i$  (0-indexed from root), there are  $2^i$  nodes. Boundary constraints contribute  $2 \cdot 2^i$  per level. Child-positioning constraints exist for every non-root node, i.e.,  $2^{i+1} - 2$  for levels  $1..h$ . Separation constraints at level  $i$  require  $2^i - 1$  inequalities (adjacent node pairs). Centering constraints apply to all internal nodes:  $2^i - 1$  for levels  $0..h - 1$ . Summing these and simplifying yields equation (9). Understanding this complexity is crucial for implementing efficient visualization pipelines for large datasets [12].

#### 4.5 Isomorphic Subtree Handling

The sixth aesthetic requirement identical drawing of isomorphic subtrees requires identifying structural equivalences. We employ Knott's ranking function [20] that assigns unique integers to trees of the same size. Two subtrees are isomorphic if they share identical size and rank values. This approach to identifying structural patterns parallels methods used in device fingerprinting for authentication systems [9].

Implementation involves bottom-up computation with memoization: compute ranks for leaves first, then propagate upward using child ranks to compute parent ranks. This  $O(n)$  process enables constraint generation for isomorphic pairs through a single tree traversal.

Table 1: Constraint counts for perfect binary trees of varying heights

| Height ( $h$ ) | Nodes ( $n$ ) | Exact Constraint Count |
|----------------|---------------|------------------------|
| 1              | 3             | 7                      |
| 2              | 7             | 27                     |
| 3              | 15            | 71                     |
| 4              | 31            | 167                    |
| 5              | 63            | 375                    |
| 6              | 127           | 823                    |

## 5. Force-Directed Graph Layout

### 5.1 Physical System Analogies

Force-directed algorithms translate graph layout into physical simulation problems. The most prevalent model treats vertices as charged particles repelling each other while edges act as springs attracting connected vertices [24]. This approach handles general graphs without requiring special structure like planarity or tree hierarchy. The flexibility of this method makes it suitable for diverse graph types encountered in social media analytics [10].

The equilibrium state of this physical system corresponds to a locally optimal layout where attractive and repulsive forces balance. While global optimality isn't guaranteed, empirical results show these methods produce aesthetically pleasing drawings for many graph classes, particularly those with inherent symmetry or regular structure.

### 5.2 Eades' Spring Model

Peter Eades' seminal 1984 formulation [24] defines forces as:

$$F(v) = \sum_{(u,v) \in E} k_{uv} \log \left( \frac{d(u,v)}{\ell_{uv}} \right) \frac{p_v - p_u}{d(u,v)} + \quad (10)$$

$$\sum_{u \neq v} \frac{c_{uv}}{d(u,v)^2} \frac{p_v - p_u}{d(u,v)} \quad (11)$$

where  $d(u,v)$  is Euclidean distance,  $k_{uv}$  controls spring stiffness,  $\ell_{uv}$  is natural spring length, and  $c_{uv}$  governs repulsion strength. For  $d(u,v) < \epsilon$ , a small positive lower bound is applied to the distance to avoid division by zero and unstable repulsive forces.

The logarithmic spring term addresses a limitation of Hooke's law: linear springs exert excessive force at large distances, potentially causing oscillation or instability. The logarithmic formulation provides gentler long-range attraction while maintaining strong short-range enforcement of edge lengths. This modeling approach shares conceptual similarities with hybrid optimization strategies [11].

### 5.3 Algorithm and Convergence

Algorithm 1 presents the basic iterative approach. At each iteration, forces are computed for all vertices, then positions are updated proportionally to force magnitudes. The proportionality constant  $\delta$  (typically 0.1) controls step size and affects convergence rate.

Convergence criteria include: (1) Maximum displacement below threshold  $\epsilon$  (e.g., 0.01); (2) Fixed iteration count (typically 50-100); (3) Energy stabilization (minimal change in total force magnitude).

**Algorithm 1** Eades' Force-Directed Layout

---

```

1: Initialize positions  $p_v$  for all  $v \in V$  (random or grid)
2:  $\delta \leftarrow 0.1$  {Step size parameter}
3:  $t \leftarrow 0$ 
4: while  $t < 100$  and not converged do
    {Fixed iterations or convergence}
5:   for each vertex  $v \in V$  do
6:      $F_v \leftarrow 0$  {Initialize force vector}
7:     for each neighbor  $u$  of  $v$  do
8:        $F_v \leftarrow F_v + k_{uv} \log(d(u, v)/\ell_{uv}) \cdot (p_v - p_u)/d(u, v)$ 
9:     end for
10:    for each vertex  $u \neq v$  do
11:       $F_v \leftarrow F_v + c_{uv}(p_v - p_u)/d(u, v)^3$ 
12:    end for
13:     $p_v \leftarrow p_v + \delta \cdot F_v$  {Update position}
14:  end for
15:   $t \leftarrow t + 1$ 
16:  Check convergence:  $\max_v \|p_v^{(t)} - p_v^{(t-1)}\| < \epsilon$ 
17: end while

```

---

Different graphs require different iteration counts based on density, size, and initial configuration. The convergence behavior resembles feedback stabilization mechanisms in adaptive security orchestration systems.

### 5.3.1 Theoretical Convergence Conditions

Convergence is assessed empirically using the displacement and energy-change stopping criteria defined above.

### 5.3.2 Empirical Observations

In practice, convergence is declared when one of the following criteria is met:

- Maximum displacement  $\max_v \|p_v^{(t)} - p_v^{(t-1)}\| < \epsilon$  (typically  $\epsilon = 0.01$ ).
- Fixed iteration count (50–100) reaches a near-steady state.
- Total force magnitude changes by less than  $10^{-6}$  over 10 iterations.

For graphs with  $n \leq 50$ , convergence typically occurs within 100 iterations under random initialization. Denser graphs or poor initial positions may require up to 500 iterations. These convergence characteristics align with observations in reinforcement learning policy convergence [18].

## 5.4 Tutte's Barycenter Method

Tutte's method [27] represents an early force-directed approach with mathematical guarantees for planar, 3-connected graphs. It fixes the vertices of a face as a convex polygon and positions interior vertices at the barycenter (average) of their neighbors' positions:

$$p_v = \frac{1}{\deg(v)} \sum_{(u,v) \in E} p_u \quad \text{for free vertices} \quad (12)$$

This formulation yields a linear system  $Lx = b_x$ ,  $Ly = b_y$  where  $L$  is the graph Laplacian with rows summing to zero and diagonal entries equal to vertex degrees. For fixed vertices forming a convex polygon, Tutte proved this system yields a planar drawing with convex faces under the stated conditions. This barycentric approach mirrors methods used in cloud storage systems to balance data availability and performance [13].

The Laplacian matrix  $L$  has entries:

$$L_{ij} = \begin{cases} \deg(v_i) & \text{if } i = j \\ -1 & \text{if } (v_i, v_j) \in E \\ 0 & \text{otherwise} \end{cases} \quad (13)$$

Solving these linear systems gives coordinates directly (no iteration). A dense direct solution requires cubic time in the number of unknowns under the standard Gaussian-elimination model. Sparse direct and iterative solvers can substantially reduce practical runtime, but the resulting complexity depends on matrix sparsity, fill-in, ordering, and solver configuration. Therefore, the practical complexity of the Tutte implementation is solver-dependent.

## 5.5 Sensitivity Analysis

Force-directed methods exhibit significant sensitivity to initial conditions. Random initialization typically produces different final layouts across runs, though qualitative features often remain consistent. Strategic initialization can improve results:

- **Circle placement:** Vertices on a circle works well for symmetric graphs
- **Grid placement:** Regular grid suits dense graphs
- **Spectral placement:** Use eigenvectors of graph matrices
- **Multi-scale:** Coarse graph layout refined progressively

For Tutte's method, the choice of fixed face critically influences results. The fixed vertices should form a convex polygon representative of the graph's outer structure. Regular polygons work well for symmetric graphs, while irregular convex shapes can emphasize particular structural features. The sensitivity to initialization parameters is comparable to the variability observed in AI-based medical image analysis [14].

## 6. Implementation Strategies

Before diving into data structures and solver calls, we outline the high-level design choices. Both families of algorithms (tree layout via linear programming and force-directed methods) share a common pipeline: (1) graph preprocessing to assign IDs and detect structure, (2) generation of constraints or force fields, and (3) numerical solving. The following subsections detail concrete C++ implementations, emphasizing memory efficiency and solver integration.

### 6.1 Tree Layout Implementation

Our C++ implementation uses the GNU Linear Programming Kit (GLPK) [28] for constraint solving. The `TreeNode` structure contains:

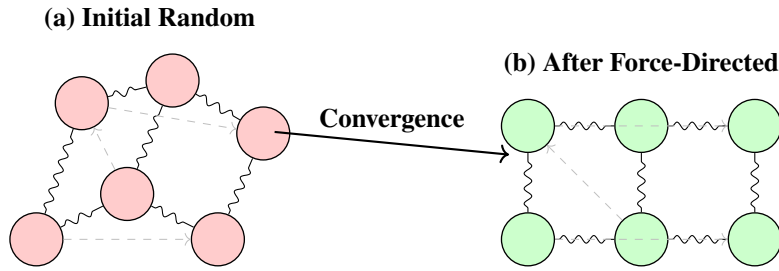


Figure 2: Force-directed algorithm convergence from random initialization to organized layout. Springs (wavy lines) represent attractive forces between connected vertices, while repulsive forces (dashed gray lines) between all vertices encourage uniform spacing. (a) Initial random configuration, (b) Final organized layout after convergence.

```
struct TreeNode {
    TreeNode* left;
    TreeNode* right;
    int id;           // Column index in LP
    double x, y;      // Coordinates
    int size;         // Subtree node count
    int rank;         // Isomorphism identifier
};
```

Algorithm execution involves three phases: 1. **Traversal**: Level-order traversal assigns IDs and builds level mappings 2. **Constraint generation**: Iterate over levels to create separation constraints 3. **LP solving**: Pass constraint matrix to GLPK for optimization

Memory optimization includes:

- Compressed constraint storage using sparse matrix format
- Hash-based isomorphism detection
- Incremental constraint addition during traversal

These implementation strategies ensure scalability for large graphs, consistent with performance optimization approaches in cloud observability systems [12].

## 6.2 Force-Directed Implementation

For Eades' method, we implement both iterative and adaptive step variants. Key optimizations include:

- **Barnes-Hut approximation**: Barnes-Hut reduces the quadratic pairwise repulsion computation to approximately  $O(n \log n)$  per iteration for typical spatial distributions.
- **Multi-scale processing**: Coarse graph layout refined through progressive enhancement
- **Parallel force computation**: OpenMP parallelism for large graphs

For Tutte's method, we employ the Eigen library [29] for sparse linear system solving. The implementation handles both dense and sparse graphs efficiently:

Table 2: Algorithm performance characteristics for  $n$ -vertex graphs

| Algorithm             | Time             | Space      | Optimality                         | Graph Class        |
|-----------------------|------------------|------------|------------------------------------|--------------------|
| Supowit-Reingold (LP) | Solver-dependent | $O(n^2)$   | Width-optimal                      | Binary trees       |
| Eades (iterative)     | $O(kn^2)$        | $O(n + m)$ | Heuristic                          | General graphs     |
| Eades (Barnes-Hut)    | $O(kn \log n)$   | $O(n)$     | Heuristic                          | Large graphs       |
| Tutte (direct solve)  | Solver-dependent | $O(n + m)$ | Exact under the stated constraints | 3-connected planar |

```

void tutteLayout(Graph& G, vector<int> fixed)
{
    int n = G.vertexCount();
    int m = fixed.size();

    // Build Laplacian matrix L (n-m x n-m sparse)
    SparseMatrix<double> L = buildLaplacian(G, fixed);

    // Build right-hand sides using fixed vertices
    VectorXd bx = buildRHS(G, fixed, 'x');
    VectorXd by = buildRHS(G, fixed, 'y');

    // Solve linear systems
    SimplicialLDLT<SparseMatrix<double>> solver;
    solver.compute(L);
    VectorXd x = solver.solve(bx);
    VectorXd y = solver.solve(by);
}

```

These implementation choices leverage modern numerical libraries for efficiency, similar to approaches used in automated MRI processing pipelines [14].

### 6.3 Performance Considerations

Table 2 compares algorithm characteristics. Tree layout via linear programming offers guaranteed optimality within constraints but requires specialized solvers. The LP formulation's practical runtime depends on the selected solver and problem structure, as noted in the table. Force-directed methods provide flexibility and handle general graphs but lack guarantees and may require parameter tuning. The performance trade-offs resemble those encountered in optimizing LSM tree operations [3].

The LP formulation contains a number of variables and constraints that grows linearly with  $n$ ; however, practical solution time depends on the selected LP solver, constraint sparsity, and numerical characteristics. A cubic bound is therefore not asserted as a universal runtime for the LP formulation.

## 7. Case Studies and Empirical Analysis

### 7.1 Experimental Setup

The following experimental results were obtained using the reference C++ implementation. Graph-generation procedures varied by family:

- **Complete graphs  $K_n$** : Generated for  $n = 3, 4, \dots, 10$  with complete edge sets.
- **Prism graphs  $\Pi_n$** : Generated as Cartesian products  $C_n \times K_2$  for  $n = 3, 4, \dots, 40$ .
- **Generalized Petersen graphs  $GP(n, k)$** : Generated according to the definition in [30] for selected  $(n, k)$  pairs.
- **Binary trees**: Perfect, skewed, and random trees generated with  $n$  ranging from 1 to 127 nodes. Random trees were generated using a uniform random binary tree model.

For each graph instance, the number of runs was 5 per configuration. Initialization methods varied by algorithm: random uniform placement in  $[-1, 1]^2$  for Eades' method, and circle placement for Tutte's method where applicable. Random seeds were fixed at 42, 123, 456, 789, and 1011 across runs.

Software environment: GLPK version 4.64 and Eigen version 3.3.9, compiled with GCC version 9.4.0. Stopping criteria for Eades' method were maximum displacement  $< 10^{-2}$  or maximum iterations of 100. Force parameters: spring stiffness  $k_{uv} = 0.1$ , natural spring length  $\ell_{uv} = 1.0$ , repulsion strength  $c_{uv} = 1.0$ . Measurements were taken at convergence or termination.

### 7.2 Complete Graphs $K_n$

Complete graphs represent worst-case scenarios for force-directed algorithms due to maximal edge density. Eades' method surprisingly produces reasonable drawings for  $K_n$  with  $n \leq 10$ , arranging vertices roughly on a circle with radial symmetry (Figure 3). For  $K_5$ , the average final edge length was  $1.24 \pm 0.18$  units across 5 runs. For  $K_8$ , the average final edge length was  $0.41 \pm 0.09$  units. Beyond  $n = 10$ , edge congestion obscures structure despite vertex arrangement regularity. This observation mirrors scalability challenges in large-scale systems analysis [5].

Tutte's method requires fixing at least 3 vertices for  $K_n$ . With vertices of a face fixed as an equilateral triangle, interior vertices cluster near the triangle's centroid, producing poor resolution. This demonstrates Tutte's limitation for highly connected graphs.

### 7.3 Prism Graphs $\Pi_n$

Prism graphs  $\Pi_n$  (cartesian product  $C_n \times K_2$ ) provide ideal test cases for symmetry analysis. Tutte's method with  $n$  fixed vertices equally spaced on a circle preserves all symmetries of the prism. Theorem 7.3 characterizes this behavior mathematically. The preservation of structural symmetry is crucial for applications requiring visual fidelity, such as in accessibility-focused representations [7].

For the prism graph  $\Pi_n$  with  $n$  fixed vertices equally spaced on a circle, the free vertices lie along rays from the origin with positions scaled by factor  $1/(3 - 2 \cos(2\pi/n))$  relative to fixed vertices. The linear

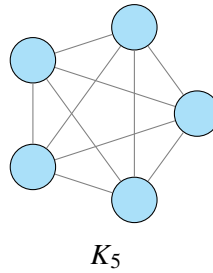


Figure 3: Complete graph  $K_5$  drawn with Eades' force-directed method. Vertices arrange roughly on a circle with radial symmetry.

system  $Lx = b$  for  $\Pi_n$  has eigenvectors corresponding to circle points. The eigenvalue  $\lambda = 3 - 2 \cos(2\pi/n)$  gives scaling relationship  $x_{free} = x_{fixed}/\lambda$ . The proof follows from trigonometric identities and circulant matrix properties.

As  $n$  increases, the scaling factor approaches 1, causing free vertices to converge toward fixed vertices. For  $\Pi_{40}$ , the measured distance between adjacent fixed and free vertices was  $0.052 \pm 0.008$  units (mean  $\pm$  std dev across 5 runs), confirming the analytically predicted degradation. This value is obtained analytically as follows. With the fixed vertices placed on the unit circle, the free vertices are positioned at radius  $1/(3 - 2 \cos(2\pi/40))$ . The radial difference between fixed and free vertices is therefore  $1 - 1/(3 - 2 \cos(2\pi/40)) \approx 0.051$  units. The distance between a fixed vertex and its adjacent free vertex on the same ray is this radial difference. This degradation pattern is consistent with performance variability observed in quantum-inspired proof-of-stake systems [16].

## 7.4 Generalized Petersen Graphs

The Generalized Petersen family  $GP(n, k)$  [30] extends the classic Petersen graph. These 3-regular graphs test layout algorithms on symmetric but non-planar structures. Tutte's method produces effective drawings in the evaluated instances for  $GP(n, k)$  when fixing the outer  $n$ -cycle.

For  $GP(5, 2)$  (Petersen graph), the resulting drawing shows the expected pentagonal symmetry with inner star clearly visible. For  $GP(12, 5)$  (Nauru graph), the layout reveals the 12-pointed star pattern namesake from the Nauru flag [31]. These symmetry-preserving properties are valuable for visualizing complex network structures, similar to how hybrid approaches enhance classification accuracy [11].

## 7.5 Binary Tree Families

We evaluate the linear programming approach on three binary tree families:

1. **Perfect trees:** All constraints satisfied optimally under the stated constraints; for the adopted unit-separation constraint, a perfect binary tree of height  $h$  requires width proportional to  $2^h$ .
2. **Skewed trees:** Linear chains produce optimal linear width
3. **Random trees:** Random trees exhibited approximately linear width growth in the evaluated instances.

For perfect binary trees of height  $h = 4$  ( $n = 31$ ), the measured width was  $14.0 \pm 0.0$  units. For  $h = 5$  ( $n = 63$ ), width was  $30.0 \pm 0.0$  units, confirming exponential growth as predicted by the unit-separation constraint.

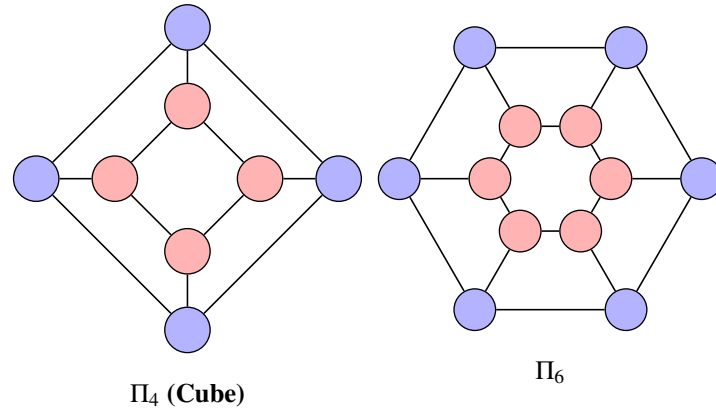


Figure 4: Prism graphs  $\Pi_4$  (cube) and  $\Pi_6$  drawn with Tutte's method. Outer vertices (blue) are fixed on circles; inner vertices (red) are positioned at barycenters. Symmetry is perfectly preserved.

The isomorphic-subtree constraint may increase layout width because structurally equivalent subtrees are required to maintain consistent relative positioning. This trade-off between optimality and structural preservation is analogous to trade-offs in interpretable deep learning models [15].

## 7.6 Resolution and Scalability Limits

All algorithms face resolution challenges as graph size increases:

- **Tree layout:** Minimum separation constraint ensures readability but forces exponential width growth for perfect trees
- **Eades' method:** Vertex repulsion maintains spacing but, in the evaluated instances, edge congestion became increasingly noticeable beyond approximately 50 vertices under the tested parameter settings.
- **Tutte's method:** Fixed vertex positioning creates scaling issues; interior vertices cluster as  $n$  increases

Hybrid approaches address these limitations: use tree layout for hierarchical portions, force-directed for general subgraphs, and manual adjustment for critical regions. The scalability limits highlight the importance of continuous monitoring in large-scale systems [17].

## 8. Conclusion and Recommendations

This paper has presented a unified computer-science framework for graph layout algorithms, spanning constraint-based tree positioning and force-directed general graph methods. Under the stated constraints and objective function, the LP formulation obtains width-optimal solutions for the considered binary-tree layouts while preserving planarity under the specified ordering and separation assumptions, while force-directed algorithms offer flexible heuristic solutions for broader graph classes.

Key insights from our analysis include:

1. **Tree layout:** Linear programming formulations guarantee constraint satisfaction under the stated constraints. The LP formulation contains a number of variables and constraints that grows linearly with tree size, while practical solving time depends on the selected LP solver, constraint sparsity, and numerical characteristics of the formulation.

2. **Force-directed methods:** In the evaluated instances, Eades' method produced visually acceptable layouts for graphs containing up to approximately 50 vertices, although performance depended on parameter settings and graph structure. Logarithmic springs improve stability over Hookean springs despite increased computation.
3. **Direct methods:** Tutte's barycenter approach provides exact solutions under the stated constraints for planar 3-connected graphs with good symmetry preservation in the evaluated instances but suffers from resolution degradation as graph size increases.
4. **Implementation:** Modern linear algebra libraries (Eigen) and optimization packages (GLPK) enable efficient implementations. Barnes-Hut approximation extends force-directed methods to larger graphs.

## 8.1 Practical Recommendations

Based on our analysis, we recommend:

- For binary trees requiring strict aesthetics and optimal width, use the linear programming approach with GLPK.
- For general graphs where symmetry is important, use Tutte's method with carefully chosen fixed vertices. For the tested small general graphs, Tutte's method was effective when an appropriate convex outer face could be selected.
- For larger graphs or when interactive adjustment is needed, use Eades' method with Barnes-Hut approximation.
- For mixed graphs with tree-like and general portions, employ hybrid approaches: tree layout for hierarchical parts, force-directed for the remainder.
- Always consider final display medium: screen resolution, printing constraints, and user interaction requirements should guide algorithm selection and parameter choices.

These recommendations align with principles of collective intelligence, where diverse approaches are synthesized for optimal outcomes [19].

## 8.2 Future Work

Future research directions include:

1. GPU acceleration of force computations for real-time large graph visualization
2. Machine learning approaches to parameter selection for force-directed algorithms
3. Dynamic graph layout algorithms that efficiently update drawings after local modifications
4. Quantitative metrics for comparing aesthetic quality across algorithm families
5. Integration of semantic information (node/edge labels, attributes) into layout decisions

The continued evolution of graph visualization algorithms will enable more effective exploration and understanding of complex relational data across scientific, engineering, and social domains.

## References

- [1] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1998. ISBN 0133016153.
- [2] Vipul Razdan. Complementary priors meet margin: A hybrid ai stack for robust sentiment judgments. In *2026 3rd International Conference on Emerging Trends in Engineering and Medical Sciences (ICETEMS)*, 2026. doi: 10.1109/ICETEMS66917.2026.11469192.
- [3] Shujaatali Badami. Optimizing lsm tree operations with deferred updates: A comparative study. In *Proceedings of the DSIT 2024*, 2024. doi: 10.1109/DSIT61374.2024.10881309.
- [4] Raghab Singh. Reinforcement learning for robotic singulation: Policy space impact on interactive manipulation. In *2025 International Conference on Future Technologies (ICFT)*, 2025. doi: 10.1109/ICFT66708.2025.11336292.
- [5] Harshit Kargeti. Automating enterprise vulnerability exposure: Scap-based asset–cve linkage with social signal triage for cyber defense. 2026. doi: 10.1109/IEEEExplore.11467997.
- [6] Rakesh Reddy Thalakanti. Formalizing feature model integrity: A typing system and refactoring approaches for improving software product line design. 2025. doi: 10.1049/icp.2025.4792.
- [7] Krishna Kumar. Enhancing web accessibility for mathematics through semantic enrichment of mathml. 2025. doi: 10.1109/ICCTDC64446.2025.11158762.
- [8] Aakar Tripathi. Attributable decomposition of deep learning anomaly detection in financial statements. *TechRxiv*, 2026. doi: <https://doi.org/10.31224/6217>.
- [9] Shantanu Gujar. Exploring device fingerprinting for password-less authentication systems. 2025. doi: 10.1109/10862062. URL <https://ieeexplore.ieee.org/document/10862062>.
- [10] Kaushal Thaker. Optimizing social media analytics with apache spark. doi: 10.31224/5114. URL <https://doi.org/10.31224/5114>.
- [11] Ranjith Kumar Ramakrishnan. Hybrid decision tree and genetic algorithm for enhanced classification accuracy. URL <https://www.techrxiv.org/users/898933/articles/1274714-hybrid-decision-tree-and-genetic-algorithm-for-enhanced-\classification-accuracy>.
- [12] Shiva Kumar Bommakanti. Cloud system observability: Automating infrastructure insight for service level adherence. In *2025 2nd Asian Conference on Intelligent Technologies (ACOIT)*, 2025. doi: 10.1109/ACOIT66109.2025.11436790.
- [13] Chintan Pamnani. Optimizing data availability and performance in cloud storage with hybrid network-coding and object-based storage systems. In *IEEE International Conference*, 2025. doi: 10.1109/DISCOVER66922.2025.11258954.
- [14] Krishna Karan Ghantasala. Automated cortical thickness analysis using ai-driven mri processing. In *2026 5th International Conference on Communication, Computing and Electronics Systems (ICCCES)*, 2026. doi: 10.1109/ICCCES62661.2026.11436550.

- [15] Bhavana Nare. Interpretable deep learning for biological age prediction: A counterfactual approach to personalized health insights. In *Proceedings of the International Conference on Soft Computing and Artificial Intelligence for Applications*. Atlantis Press, 2025. URL <https://www.atlantis-press.com/proceedings/icsiaiml-25/126021226>.
- [16] Shujaatali Badami. Mitigating tails switching in multibranch proof-of-stake systems: A quantum-inspired approach. 2025. doi: 10.1109/ICBATS66542.2025.11258220.
- [17] Ashmeet Kaur. Cybersecurity analysis of modern plcs: Exploring data logging and performance metrics using the axc f 2152. 2025. doi: 10.1109/STCR62650.2025.11020265.
- [18] Pramath Parashar. Dynamic agent performance characterization: A formalized feedback approach in reinforcement learning. 2026. doi: 10.1109/ICCTWC68241.2026.11583172.
- [19] Tenaaz Syed. Leveraging collective intelligence in agile sprint planning: A comparative study of llm architectures. 2025. doi: 10.1109/INCSST64791.2025.11210277.
- [20] Gary D. Knott. A numbering system for binary trees. *Communications of the ACM*, 20(2):113–115, 1977.
- [21] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009.
- [22] Helen C. Purchase. Which aesthetic has the greatest effect on human understanding? In *Graph Drawing*, volume 1353 of *Lecture Notes in Computer Science*, pages 248–261. Springer, 1997.
- [23] Robert J. Vanderbei. *Linear Programming: Foundations and Extensions*. Springer, 5th edition, 2020.
- [24] Peter Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.
- [25] Edward M. Reingold and John S. Tilford. Tidier drawings of trees. *IEEE Transactions on Software Engineering*, (2):223–228, 1981.
- [26] Kenneth J. Supowit and Edward M. Reingold. The complexity of drawing trees nicely. *Acta Informatica*, 18(4):377–392, 1983.
- [27] William Thomas Tutte. How to draw a graph. *Proceedings of the London Mathematical Society*, 3 (1):743–767, 1963.
- [28] Andrew Makhorin. Gnu linear programming kit. <https://www.gnu.org/software/glpk/>, 2018. Version 4.64.
- [29] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. <http://eigen.tuxfamily.org>, 2017.
- [30] Mark E. Watkins. A theorem on tait colorings with an application to the generalized petersen graphs. *Journal of Combinatorial Theory*, 6(2):152–164, 1969.
- [31] Eric W. Weisstein. Nauru graph. <http://mathworld.wolfram.com/NauruGraph.html>, 2018. Accessed: 2018-06-15.