Line Graphs: Structure, Recognition, and Inversion

Clement Wang

2025–2026

These notes develop the theory of line graphs from first principles, with some proofs, for a reader with a mathematical background. After the graph-theoretic preliminaries, we define the line graph $L(G)$, study it as an operator (injectivity, the triangle/claw exception, non-surjectivity), and give the classical characterizations and inversion algorithms. We close with a survey of how line graphs are used in machine learning, from link prediction to community detection.

Contents

1 Introduction and Motivation **3**
- 1.1 The Edge–Vertex Duality . . . 3
- 1.2 Why Inversion and Modification Matter? . . . 3
- 1.3 Roadmap . . . 3

2 Graph-Theoretic Preliminaries **3**
- 2.1 Basic Definitions and Notation . . . 3
- 2.2 Special Families and Induced Subgraphs . . . 4
- 2.3 Traversals: Eulerian and Hamiltonian . . . 5

3 The Line Graph: Definition and Basic Invariants **6**
- 3.1 Definition and Examples . . . 6
- 3.2 Order, Size, and Degrees of the Line Graph . . . 7
- 3.3 Traversal Correspondences . . . 8

4 The Line Graph Operator **9**
- 4.1 The Exceptional Pair: Triangle and Claw . . . 9
- 4.2 Whitney's Isomorphism Theorem (Injectivity) . . . 9
- 4.3 Non-Surjectivity: Not Every Graph Is a Line Graph . . . 10
- 4.4 Krausz's Clique-Partition Characterization . . . 10
- 4.5 Local Structure: Odd and Even Triangles . . . 10
- 4.6 Beineke's Forbidden-Subgraph Theorem . . . 11
- 4.7 Summary: Three Equivalent Characterizations . . . 12

5 Inverting the Line Graph: Root Graph Reconstruction **12**
- 5.1 The Reconstruction Problem . . . 12
- 5.2 Roussopoulos's Algorithm . . . 13
- 5.3 Lehot's Algorithm . . . 13

6 Line Graph Modification Problems **14**

7 Line Graphs and Machine Learning **15**
7.1 Link Prediction on the Line Graph 15
7.2 Community Detection via the Line Graph 15
7.3 Combining a Graph and Its Line Graph: ncRNA–Protein Interaction 15

8 Conclusion **15**

1 Introduction and Motivation

1.1 The Edge–Vertex Duality

Given a graph $G = (V, E)$, its *line graph* $L(G)$ is obtained by a simple change of perspective: the *edges* of G become the *vertices* of $L(G)$, and two of these new vertices are joined whenever the corresponding edges of G share an endpoint. Adjacency of edges in G thus becomes adjacency of vertices in $L(G)$.

This dualization is more than a notational trick. A surprising number of questions about the edges of G translate into questions about the vertices of $L(G)$. For example, demonstrating coloring properties of edges of G translates into demonstrating coloring properties of vertices of $L(G)$. The line graph is the natural transformation for moving structure between the two levels.

1.2 Why Inversion and Modification Matter?

The map $G \mapsto L(G)$ raises three questions:

- **Is it injective?** If $L(G) \cong L(H)$, must $G \cong H$? Almost always yes: Whitney's theorem says the only exception is a single small pair, the triangle and the claw.
- **Is it surjective?** Is every graph a line graph? No: there are several characterisations of line graphs. One of them is a finite list of nine forbidden patterns (Beineke).
- **Can it be inverted or approximately inverted in reasonable time?** Determining whether a graph is a line graph and recovering the root graph is doable in polynomial time. However, the problem is much harder if the graph is not a line graph.

1.3 Roadmap

Section 2 fixes the graph-theoretic vocabulary: special families, induced subgraphs, and the Eulerian/Hamiltonian traversal theory. Section 3 defines the line graph and computes its basic invariants. Section 4 studies the operator itself (injectivity, non-surjectivity, and the three classical characterizations of line graphs: Krausz, odd/even triangles, Beineke). Section 5 presents the linear-time inversion algorithms, and Section 6 the edge-modification problems. Section 7 gives some applications of line graphs in machine learning.

2 Graph-Theoretic Preliminaries

2.1 Basic Definitions and Notation

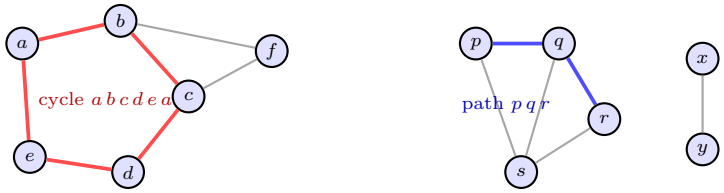
Definition 2.1 (Graph)**.** A *(simple, undirected) graph* is a pair $G = (V, E)$ where V is a finite set of *vertices* and $E \subseteq \binom{V}{2}$ is a set of *edges*, each an unordered pair $\{u, v\}$ of distinct vertices. We write $n = |V|$ for the *order* and $m = |E|$ for the *size*. An edge $e = \{u, v\}$, written uv for short, is *incident* to u and v, which are its *endpoints*; u and v are then *adjacent*. Unless stated otherwise all graphs are finite, simple (no loops or multiple edges), and undirected.

Definition 2.2 (Neighborhood and degree)**.** The *neighborhood* of a vertex v is the set of vertices adjacent to it, $\mathcal{N}(v) = \{u \in V : uv \in E\}$. Its *degree* is the size of that set, $d_v = |\mathcal{N}(v)|$.

Definition 2.3 (Walk, trail, path)**.** A *walk* is an alternating sequence of vertices and edges $v_0\, e_1\, v_1\, e_2 \cdots e_k\, v_k$ with $e_i = v_{i-1}v_i \in E$; its *length* is the number of edges k. A walk that never reuses an edge is a *trail*. A walk that never revisits a vertex is a *path*. Every path is a trail, since repeating a vertex is the only way to repeat an edge without first repeating a vertex.

Definition 2.4 (Closed walk, circuit, cycle)**.** A walk is *closed* when it returns to its start, $v_0 = v_k$. A closed trail is a *circuit*. A closed walk whose only repeated vertex is the endpoint $v_0 = v_k$, with $k \geq 3$, is a *cycle*; the cycle on k vertices is C_k.

Definition 2.5 (Connectivity, components, distance, diameter)**.** G is *connected* if some walk joins every pair of vertices. The maximal connected subgraphs of G are its *components*. When u and v lie in the same component, their *distance* $\operatorname{dist}(u, v)$ is the length of a shortest u–v path; the largest distance between any two vertices is the *diameter* $\operatorname{diam}(G) = \max_{u,v} \operatorname{dist}(u, v)$.



one component, diam = 2 two components

Figure 1: Left: a connected graph; the red edges trace the cycle $a\,b\,c\,d\,e\,a$, and the longest shortest-path distance is 2 (e.g. $\operatorname{dist}(a, f) = 2$). Right: a graph with two components; the blue edges trace the path $p\,q\,r$, while x and y sit in a separate component, so $\operatorname{dist}(p, x)$ is undefined.

Lemma 2.6 (Handshake lemma)**.** *For every graph,* $\sum_{v \in V} d_v = 2m$.

Proof. Each edge uv contributes exactly 1 to d_u and 1 to d_v, hence 2 to the sum $\sum_v d_v$; summing over all m edges gives $2m$. □

Remark 2.7 (Standing conventions). We work up to isomorphism: $G \cong H$ means there is a bijection $V(G) \to V(H)$ preserving adjacency, and we identify graphs that are isomorphic whenever only their structure matters. Graphs are assumed connected with at least one edge unless noted, which avoids degenerate special cases in the line-graph theory.

Notation summary

Symbol	Meaning	Reference
$G = (V, E)$	simple undirected graph	Def. 2.1
$n = \lvert V\rvert,\ m = \lvert E\rvert$	order and size	Def. 2.1
$d_v,\ \mathcal{N}(v)$	degree and neighborhood of v	Def. 2.2
$G[S]$	subgraph induced by $S \subseteq V$	Def. 2.14
$L(G)$	line graph of G	Def. 3.1
K_n	complete graph / clique on n vertices	Def. 2.8
$P_n,\ C_n$	path and cycle on n vertices	Def. 2.9
$K_{1,n}$	star with n leaves	Def. 2.10
$K_{1,3}$	claw (the star $K_{1,3}$)	Def. 2.10
$K_{p,q}$	complete bipartite graph	Def. 2.11
B	vertex–edge incidence matrix	Def. 3.1

2.2 Special Families and Induced Subgraphs

We first name the small graphs that recur throughout, then the subgraph relation that lets us speak of one graph sitting inside another. The two are tied together at the end: line graphs are exactly the graphs that avoid a short list of these small graphs as *induced* subgraphs.

Definition 2.8 (Complete graph / clique K_n)**.** The *complete graph* K_n has n vertices and every pair of vertices is adjacent, hence it has $\binom{n}{2}$ edges. A *clique* of a graph G is a vertex set inducing a complete subgraph.

Definition 2.9 (Path P_n and cycle C_n)**.** The *path* P_n (a *chain*) has vertices $v_1, \dots, v_n$ and edges $v_i v_{i+1}$ for $1 \le i < n$, so $n-1$ edges and two endpoints of degree 1. The *cycle* C_n ($n \ge 3$) adds the edge $v_n v_1$, giving n edges with every vertex of degree 2.

Definition 2.10 (Star $K_{1,n}$ and the claw $K_{1,3}$)**.** The *star* $K_{1,n}$ has one *center* adjacent to n pairwise non-adjacent *leaves* (n edges, all sharing the center). The case $n = 3$ is called the *claw* $K_{1,3}$: it is the smallest graph that is *not* a line graph (Corollary 4.5).

Definition 2.11 (Complete bipartite graph $K_{p,q}$)**.** $K_{p,q}$ has vertex set $A \cup B$ with $|A| = p$, $|B| = q$, and the edge set is exactly the set of pairs with one endpoint in each part (pq edges). The star is the special case $K_{1,n}$.

Remark 2.12 (A small zoo we reuse). Besides the families above we name a few fixed small graphs: the *triangle* $K_3 = C_3$; the *diamond* $K_4 - e$ (K_4 with one edge removed, i.e. two triangles sharing an edge); and the claw $K_{1,3}$.

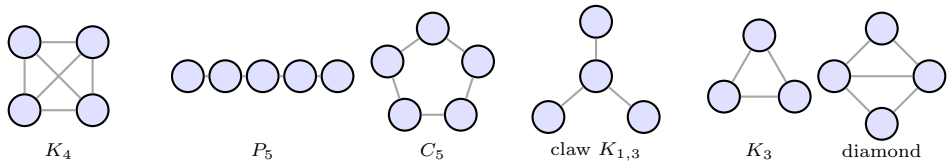


Figure 2: A gallery of small graphs

These named graphs matter mostly as patterns to look for inside a larger one, which the next two definitions make precise.

Definition 2.13 (Subgraph)**.** $H = (V', E')$ is a *subgraph* of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E \cap \binom{V'}{2}$. It is a *spanning* subgraph if $V' = V$.

Definition 2.14 (Induced subgraph, H-free graphs)**.** For $S \subseteq V$, the subgraph *induced* by S is $G[S] = (S,\ E \cap \binom{S}{2})$: it keeps *exactly* those edges of G with both endpoints in S. A graph G is *H-free* if no induced subgraph of G is isomorphic to H, and *$\mathcal{F}$-free* for a family $\mathcal{F}$ if it is H-free for every $H \in \mathcal{F}$.

Definition 2.15 (Hereditary class and forbidden subgraphs)**.** A class $\mathcal{G}$ of graphs is *hereditary* if it is closed under induced subgraphs: $G \in \mathcal{G}$ and $S \subseteq V(G)$ imply $G[S] \in \mathcal{G}$. Every hereditary class is exactly the class of $\mathcal{F}$-free graphs for $\mathcal{F}$ its set of *minimal forbidden induced subgraphs*. This family may be finite or infinite.

2.3 Traversals: Eulerian and Hamiltonian

Two ways of traversing a graph run through these notes. One asks to cross every edge once (Eulerian), the other to visit every vertex once (Hamiltonian). The line graph turns the first into the second, so we set both down here.

Definition 2.16 (Eulerian trail, circuit, graph)**.** An *Eulerian trail* is a trail using every edge of G exactly once; if it is closed it is an *Eulerian circuit*, and a connected graph possessing one is *Eulerian*.

Theorem 2.17 (Euler [1]; sufficiency due to Hierholzer [2])**.** *Let G be a connected graph with at least one edge. Then:*

(i) G has an Eulerian circuit if and only if every vertex has even degree;

(ii) G has an (open) Eulerian trail if and only if exactly two vertices have odd degree (the trail's endpoints).

Proof. Necessity. Traverse an Eulerian circuit. Each time it passes through a vertex v it uses two edges at v (one in, one out), and for a closed trail every visit is of this form; since the trail uses all edges, d_v is even. For an open trail the two endpoints are entered/left one extra time, contributing an odd count there and an even count elsewhere.

Sufficiency (Hierholzer). Suppose all degrees are even. Starting from any vertex and walking along unused edges, a trail can only get stuck by returning to its start (each intermediate vertex, having even degree, always offers an unused exit when entered); thus we obtain a circuit C. If C omits an edge, connectivity gives a vertex w on C incident to an unused edge; the graph of unused edges also has all even degrees, so the same argument yields a circuit C' through w, and we splice C' into C at w. Repeating until no edge remains produces an Eulerian circuit. The open case follows by adding a temporary edge between the two odd vertices, applying (i), and deleting it. □

Definition 2.18 (Hamiltonian path, cycle, graph)**.** A *Hamiltonian path* (resp. *cycle*) is a path (resp. cycle) visiting every vertex of G exactly once. A graph with a Hamiltonian cycle is *Hamiltonian*.

Remark 2.19 (Euler vs. Hamilton: edges vs. vertices)*.* Eulerian traversals visit every *edge* once; Hamiltonian traversals visit every *vertex* once. This is exactly the distinction the line graph exchanges: edges of G are vertices of $L(G)$, so an edge-traversal of G becomes a vertex-traversal of $L(G)$ (Proposition 3.8).

Remark 2.20 (Computational contrast)*.* Eulerianness is decidable in linear time by the degree condition of Theorem 2.17, whereas deciding Hamiltonicity is NP-complete. The line graph thus links an easy problem on G to a hard one on $L(G)$.

3 The Line Graph: Definition and Basic Invariants

3.1 Definition and Examples

Definition 3.1 (Line graph $L(G)$)**.** The *line graph* $L(G)$ of $G = (V, E)$ has vertex set $V(L(G)) = E$, and two distinct edges $e, f \in E$ are adjacent in $L(G)$ if and only if they share an endpoint in G:

$$e \sim f \quad \Longleftrightarrow \quad e \cap f \neq \varnothing.$$

Equivalently, in terms of the vertex–edge *incidence matrix* $B \in \{0,1\}^{n \times m}$ (with $B_{ve} = 1$ iff $v \in e$), the adjacency matrix of the line graph is

$$A\big(L(G)\big) = B^{\top} B - 2I_m,$$

since $(B^{\top}B)_{ef}$ counts the shared endpoints of e and f: this is 2 on the diagonal and, for $e \neq f$ in a simple graph, 1 if they meet and 0 otherwise.

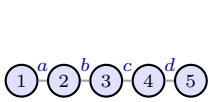
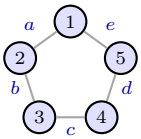
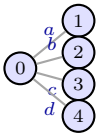
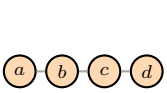
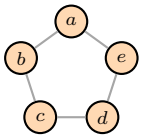
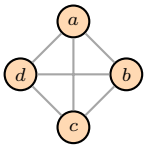
	Path P_n	**Cycle** C_n	**Star** $K_{1,n}$
G			
$L(G)$			
	$L(P_n) = P_{n-1}$	$L(C_n) = C_n$	$L(K_{1,n}) = K_n$

Figure 3: The three basic families. **Path:** the $n-1$ edges form a chain, consecutive edges sharing a vertex, so $L(P_n) = P_{n-1}$. **Cycle:** the edges again form a cycle, so C_n is a fixed point of L. **Star:** all n edges share the centre and are pairwise adjacent, giving $L(K_{1,n}) = K_n$.

Example 3.2 (Triangular graphs $L(K_n)$)**.** $L(K_n)$ is the *triangular graph* $T(n)$: its vertices are the 2-subsets of $\{1, \dots, n\}$, adjacent iff they intersect. The two smallest cases are $L(K_3) = K_3$ and $L(K_4) = K_{2,2,2}$, the octahedron (Figure 4).

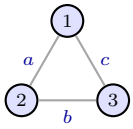
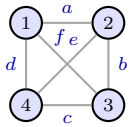
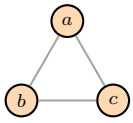
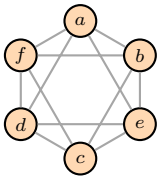
	$L(K_3) = K_3$	$L(K_4) = K_{2,2,2}$
G		
$L(G)$		

Figure 4: Triangular graphs. **Left:** K_3 has three pairwise-meeting edges, so $L(K_3) = K_3$. **Right:** the six edges of K_4 pair off into three disjoint (non-adjacent) opposite pairs, every other pair meeting; the result is the octahedron $K_{2,2,2}$.

Proposition 3.3 (Heredity)**.** *Every induced subgraph of a line graph is a line graph.*

Proof. Let $H = L(G)$ and $S \subseteq V(H) = E(G)$. Let G' be the subgraph of G formed by the edges in S together with their endpoints. For $e, f \in S$, they are adjacent in $H[S]$ iff they share an endpoint in G, iff they share an endpoint in G'; hence $H[S] = L(G')$. □

3.2 Order, Size, and Degrees of the Line Graph

Proposition 3.4 (Number of vertices)**.** $\big|V(L(G))\big| = m = |E(G)|$.

Proof. Immediate from the definition $V(L(G)) = E(G)$. □

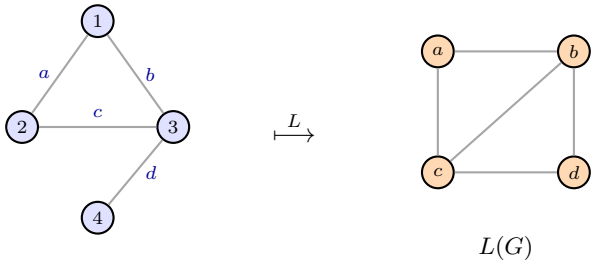


Figure 5: A graph G with edges $\{a, b, c, d\}$ and its line graph $L(G)$. Edges of G sharing an endpoint (e.g. b, c, d all meet at vertex 3) become a clique in $L(G)$.

Proposition 3.5 (Number of edges)**.**

$$\big|E(L(G))\big| \;=\; \sum_{v\in V}\binom{d_v}{2} \;=\; \frac{1}{2}\sum_{v\in V} d_v^2 \;-\; m.$$

Proof. An edge of $L(G)$ is an unordered pair of distinct edges of G sharing a common endpoint. Fix a vertex v: the edges incident to v form $\binom{d_v}{2}$ such pairs. Because G is simple, two distinct edges share *at most one* endpoint, so no pair is counted at two different vertices. Summing over v gives the first expression. For the second, expand $\binom{d_v}{2} = \frac{1}{2}(d_v^2 - d_v)$ and use $\sum_v d_v = 2m$ (Lemma 2.6). □

Proposition 3.6 (Degree of a line-graph vertex)**.** *For an edge $e = uv$ of G, its degree in $L(G)$ is*

$$d_{L(G)}(e) = d_u + d_v - 2.$$

Proof. The neighbors of e in $L(G)$ are the edges sharing an endpoint with e: the $d_u - 1$ other edges at u and the $d_v - 1$ other edges at v. These two sets are disjoint, for a common edge would have both u and v as endpoints and thus equal e. Hence $d_{L(G)}(e) = (d_u - 1) + (d_v - 1) = d_u + d_v - 2$. □

Corollary 3.7 (Regularity transfer)**.** *If G is r-regular then $L(G)$ is $(2r-2)$-regular.*

3.3 Traversal Correspondences

Proposition 3.8 (Euler in $G \Rightarrow$ Hamilton in $L(G)$)**.** *If G is connected and Eulerian, then $L(G)$ is Hamiltonian.*

Proof. Let $e_1 e_2 \cdots e_m e_1$ be an Eulerian circuit of G (Theorem 2.17), listing each edge exactly once. Read as a sequence of vertices of $L(G)$, every vertex appears exactly once. Consecutive edges e_i, e_{i+1} share an endpoint (the vertex of G between them on the circuit), so they are adjacent in $L(G)$; likewise e_m and e_1. Hence the sequence is a Hamiltonian cycle of $L(G)$. □

Remark 3.9 (The converse fails; the exact criterion). $L(G)$ can be Hamiltonian without G being Eulerian: $L(K_{1,3}) = K_3$ is Hamiltonian though $K_{1,3}$ has odd-degree vertices.

Remark 3.10 (Line graphs growth). If G is a path, $L(G)$ shrinks to a shorter path. If G is a circle, $L(G)$ stays a circle of same size. Otherwise, if a vertex has degree ≥ 3, $|V(L(G))|$ grows. Line-graph iteration thus almost always inflates the graph.

4 The Line Graph Operator

4.1 The Exceptional Pair: Triangle and Claw

The map L is not injective. A counterexample is the pair K_3 and $K_{1,3}$.

Proposition 4.1 ($L(K_3) = L(K_{1,3}) = K_3$)**.** *The triangle and the claw have isomorphic line graphs, both equal to K_3. In particular L is not injective.*

Proof. K_3 has three edges, pairwise sharing a vertex, so all three are mutually adjacent in the line graph: $L(K_3) = K_3$. The claw $K_{1,3}$ has three edges, all sharing the center, hence again pairwise adjacent: $L(K_{1,3}) = K_3$. As $K_3 \not\cong K_{1,3}$ but their images coincide, L is not injective. □

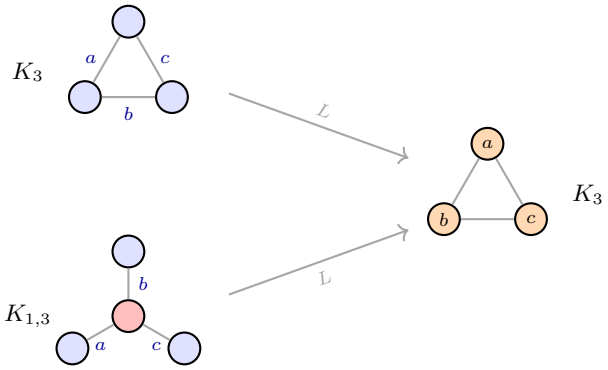


Figure 6: Both the triangle K_3 and the claw $K_{1,3}$ map to K_3

Remark 4.2 (Why exactly this pair). The coincidence is a statement about clique partitions of K_3. The single triangle $\{a, b, c\}$ of $L = K_3$ can be covered either by one clique of size 3 (one vertex of the root incident to all three edges: the claw) or by three cliques of size 2 (three vertices, each a pair-sharing: the triangle). Non-uniqueness of the Krausz partition is precisely the source of the exception, and it is the *only* graph where it happens (Section 4.4).

4.2 Whitney's Isomorphism Theorem (Injectivity)

Apart from the pair $\{K_3, K_{1,3}\}$, a connected graph is determined up to isomorphism by its line graph.

Theorem 4.3 (Whitney [3])**.** *Let G and H be connected graphs with $L(G) \cong L(H)$. Then $G \cong H$, unless $\{G, H\} = \{K_3, K_{1,3}\}$. Equivalently, L is injective up to isomorphism on connected graphs apart from the single exceptional pair $\{K_3, K_{1,3}\}$.*

Proof idea. The theorem follows once we can reconstruct G from $L(G)$. Take a vertex v of G: the edges meeting at v are pairwise adjacent in $L(G)$, so they form a clique S_v. Since an edge uv of G has just two endpoints, the corresponding vertex of $L(G)$ lies in exactly two of these cliques, S_u and S_v. So the cliques S_v cover $L(G)$ with every vertex used exactly twice.

Reversing this recovers G: take one vertex of G per clique, and join two cliques that share a vertex of $L(G)$. Whitney's theorem is that, except in finitely many small cases, the cliques S_v coincide with the maximal cliques of $L(G)$ and so are determined by $L(G)$ alone. The only ambiguity is a triangle in $L(G)$, which can arise either from three edges sharing a vertex (the claw) or from a triangle K_3 in G; this is the coincidence of Proposition 4.1. □

4.3 Non-Surjectivity: Not Every Graph Is a Line Graph

Proposition 4.4 (The claw is not a line graph)**.** *There is no graph G with $L(G) \cong K_{1,3}$.*

Proof. Suppose $L(G) \cong K_{1,3}$. The center corresponds to an edge $e = uv$ of G, and the three leaves to three edges f_1, f_2, f_3 of G, each adjacent to e but pairwise non-adjacent. Each f_i shares an endpoint with e, so it contains u or v. By the pigeonhole principle two of the three, say f_1, f_2, contain the same endpoint, say u. But then f_1 and f_2 share u, so they are adjacent in $L(G)$ which contradicts that the leaves are pairwise non-adjacent. □

Corollary 4.5 (Claw-freeness is necessary)**.** *Every line graph is claw-free ($K_{1,3}$-free). Consequently L is not surjective: $K_{1,3}$, $K_{1,4}$, and indeed every graph containing an induced claw fail to be line graphs.*

Proof. If a line graph $L(G)$ contained an induced $K_{1,3}$ on vertex set S, then by Proposition 3.3 the induced subgraph $L(G)[S] \cong K_{1,3}$ would itself be a line graph, contradicting Proposition 4.4. □

Remark 4.6 (Claw-freeness is not sufficient). Claw-freeness is necessary but *not* sufficient. The precise extra conditions are supplied by the eight further forbidden graphs of Beineke (Section 4.6).

4.4 Krausz's Clique-Partition Characterization

Definition 4.7 (Krausz partition)**.** A *Krausz partition* of a graph H is a partition of its edge set $E(H)$ into cliques such that every vertex of H lies in at most two of these cliques.

Theorem 4.8 (Krausz 1943)**.** *A graph H is a line graph if and only if it admits a Krausz partition. Moreover the cliques of the partition correspond to the vertices of a root graph G with $L(G) \cong H$.*

Proof. ($\Rightarrow$) Let $H = L(G)$. For each $v \in V(G)$ let $C_v = \{e \in E(G) : v \in e\}$ be the set of edges at v, viewed as vertices of H. Each C_v is a clique (its edges pairwise share v). An edge of H is a pair of edges e, f of G meeting at some vertex w; since G is simple they meet at *exactly one* vertex, so this pair lies in the clique C_w and in no other. Hence $\{C_v\}_{v \in V(G)}$ partitions $E(H)$. Finally each vertex $e = uv$ of H lies in exactly C_u and C_v, i.e. in two cliques. This is a Krausz partition.

($\Leftarrow$) Let $\mathcal{C}$ be a Krausz partition of H. Build G as follows: take one vertex x_C of G per clique $C \in \mathcal{C}$; for each vertex h of H lying in two cliques C, C', create the edge $x_C x_{C'}$ of G; for each vertex h lying in only one clique C, add a new degree-1 vertex and the corresponding pendant edge at x_C. This defines a bijection $V(H) \to E(G)$, $h \mapsto$ its edge. Two vertices h, h' of H are adjacent iff the pair hh' is an edge of H, which (being in the partition) lies in a unique clique C, i.e. iff h, h' both lie in C, i.e. iff their edges of G share the endpoint x_C. Two cliques meet in at most one vertex of H (otherwise two distinct edges hh' of H would lie in both, contradicting that $\mathcal{C}$ partitions $E(H)$), so the shared endpoint is well defined. Hence $H \cong L(G)$. □

Remark 4.9 (Non-uniqueness). The construction shows the root is read off directly from a Krausz partition so root graphs are in bijection with Krausz partitions. The triangle K_3 is the unique graph with two essentially different Krausz partitions, which is exactly the Whitney exception of Section 4.1. For every other connected line graph the Krausz partition, hence the root, is unique.

4.5 Local Structure: Odd and Even Triangles

Definition 4.10 (Odd and even triangles)**.** Let T be a triangle of a graph H. A vertex $w \notin T$ is *adjacent to* T in the count sense by how many of T's three vertices it neighbors. The triangle T is *even* if every vertex of H is adjacent to 0 or 2 vertices of T, and *odd* otherwise (some vertex is adjacent to exactly 1 or exactly 3 vertices of T).

Theorem 4.11 (van Rooij–Wilf 1965)**.** *A connected graph H is a line graph if and only if*

(i) H is claw-free, and

(ii) whenever two distinct odd triangles share an edge, the four vertices they span induce a K_4.

Proof idea. By Theorem 4.8, H is a line graph exactly when it has a Krausz partition, so the task is to show that (i)–(ii) are exactly what is needed to build one, and that the odd/even dichotomy tells us where to look for its cliques.

The two triangle shapes, read off in G. Suppose $H = L(G)$ and $\{e, f, g\}$ is a triangle of H, i.e. three pairwise-meeting edges of G. Two ways for that to happen: either the three share one common vertex v of G (a *star* at v), or they meet pairwise at three *distinct* vertices, forming an actual triangle uvw of G (a *root triangle*). Checking, for each shape, how many of e, f, g a fourth edge h of G can meet shows the star gives exactly the odd triangles and the root triangle gives exactly the even ones: an h through the star's centre v meets all three (h shares v with e, f, g), and an h through a leaf but not v meets exactly one; whereas an h through one of u, v, w (but not through a second, else $h \in \{e, f, g\}$) meets exactly two of the root triangle, and any other h meets none.

Necessity, by reading the Krausz partition. Condition (i) is Corollary 4.5. For (ii), recall the partition $\{C_v\}$ from the proof of Theorem 4.8: C_v collects all edges of G at v. If e, f meet at v, the edge ef of H lies only in C_v, so *every* star triangle extending ef has its third edge through v too, hence also in C_v. Two distinct odd triangles sharing the edge ef therefore contribute two such vertices $g, g' \in C_v$; since C_v is a clique, e, f, g, g' are pairwise adjacent, i.e. they induce a K_4 (ii).

Sufficiency, by rebuilding the partition. Run this backwards: declare e, f *star-linked* if they lie together in some odd triangle. Condition (ii) is exactly the closure property that stops a chain of overlapping odd triangles from producing a cluster that is only "indirectly" related – it forces every two vertices reachable by such a chain to be directly adjacent, so each maximal cluster of star-linked vertices is itself a clique. Claw-freeness caps how many such clusters (plus the pairs supplied by root triangles) can meet at a single vertex of H, at two. Assembling the clusters, the root-triangle pairs, and the leftover degree-one padding gives a Krausz partition of H, and Theorem 4.8 then produces the root graph G. □

Remark 4.12. The odd/even dichotomy is exactly the local test Lehot's algorithm uses to find the star cliques of the root graph (Section 5.3).

4.6 Beineke's Forbidden-Subgraph Theorem

Theorem 4.13 (Beineke 1970)**.** *A graph is a line graph if and only if it contains none of nine specific graphs $\{G_1, \ldots, G_9\}$ as an induced subgraph. The nine graphs are minimal non-line-graphs. They have between four and six vertices (See figure 7).*

Proof idea. The theorem splits into two parts: a *finite* forbidden family exists, and that it is *this particular* list of nine graphs.

Finiteness. Line graphs form a hereditary class (Proposition 3.3), so by Definition 2.15 they are exactly the graphs avoiding, as induced subgraphs, the *minimal* non-line-graphs: those H for which H itself fails Theorem 4.11 but every proper induced subgraph of H satisfies it. Such a failure is always *local*: either H contains an induced claw, witnessed on 4 vertices, or H has two odd triangles sharing an edge whose span is not a K_4. Minimality of H then forbids any further vertices: an extra vertex not needed to witness the failure could be deleted, and the smaller graph would still fail Theorem 4.11 (the same claw or the same bad triangle pair survives), contradicting that H is a *minimal* counterexample. So every minimal non-line-graph has a bounded number of vertices, Beineke's count gives ≤ 6, and hence there are only finitely many of them.

Identifying the nine. This bound turns the problem into a finite one, but not a short one: one must enumerate the many graphs on 4, 5, and 6 vertices, discard those that satisfy Theorem 4.11, and among the rest keep only those all of whose proper induced subgraphs are line graphs. □

Remark 4.14 (Finiteness and its consequences). Line graphs can be recognized in polynomial time by searching for an induced copy of each G_i.

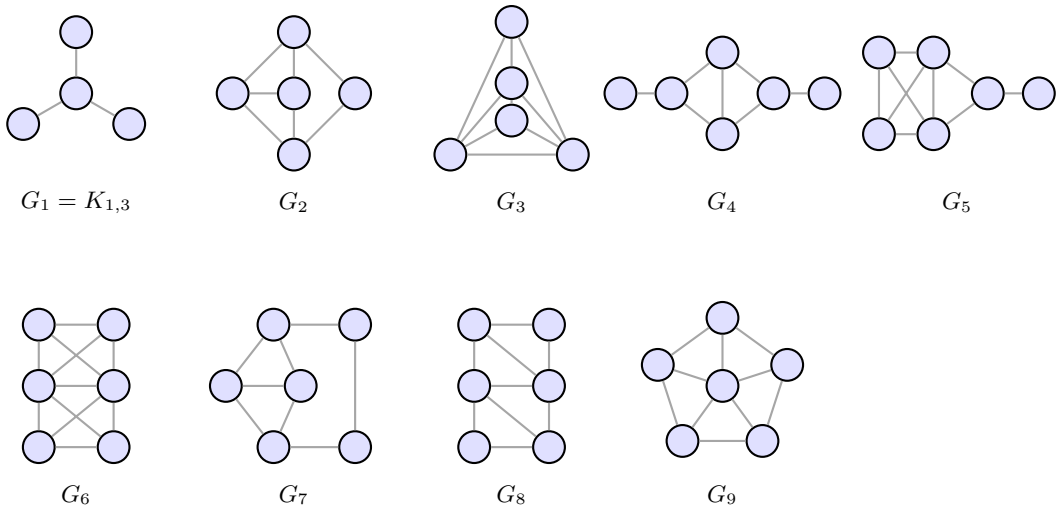


Figure 7: Beineke's nine forbidden induced subgraphs.

4.7 Summary: Three Equivalent Characterizations

Key result

For a graph H without isolated vertices, the following are equivalent:

(1) H is a line graph: $H \cong L(G)$ for some graph G;
(2) H admits a Krausz partition: its edges split into cliques with every vertex in at most two (Krausz, Theorem 4.8);
(3) H is claw-free and any two odd triangles sharing an edge span a K_4 (van Rooij–Wilf, Theorem 4.11);
(4) H is $\{G_1, \ldots, G_9\}$-free for Beineke's nine forbidden induced subgraphs (Beineke, Theorem 4.13).

5 Inverting the Line Graph: Root Graph Reconstruction

5.1 The Reconstruction Problem

Problem 5.1 (Line graph inversion / root finding)**.** Given a graph H, decide whether H is a line graph and, if so, output the root graph G, with $L(G) \cong H$.

Inverting the line graph operator gives the root which offers another view of the same graph: by Whitney's theorem it retains all of H's topological information, yet does so more compactly, with fewer edges, by collapsing each Krausz clique of H down to a single vertex of G.

By Whitney's theorem (Theorem 4.3), the root G of a connected line graph is unique up to isomorphism, with the sole exception of $H = K_3$ (whose roots are K_3 and $K_{1,3}$). The reconstruction is therefore well posed once the small case is fixed by convention.

Remark 5.2 (From Krausz to an algorithm). Theorem 4.8 reduces inversion to a purely combinatorial task: *compute a Krausz partition of H*. A valid partition immediately yields the root (one vertex per clique; each H-vertex becomes an edge between its two cliques). The two classical algorithms differ only in how they find this partition and in how they detect that none exists, certifying that H is not a line graph.

5.2 Roussopoulos's Algorithm

Roussopoulos [7] gave the first $O(n+m)$ algorithm. It builds the Krausz partition incrementally, *interleaving* recognition with construction: the moment a Krausz invariant is violated, the input is rejected as a non-line-graph.

Algorithm 1: Roussopoulos inversion (outline)

Input: a connected graph H
Output: a root graph G with $L(G) \cong H$, or REJECT
Handle base cases ($H \in \{K_1, K_2, K_3, \dots\}$) explicitly; for $H = K_3$ return the chosen convention ($K_{1,3}$ or K_3);
Pick an edge $e = \{a, b\}$ of H and let $N = N(a) \cap N(b)$ be their common neighbours, $t = |N|$;
if *N induces a clique* **then**
    $C_0 \leftarrow \{a, b\} \cup N$, of size $t + 2$ (no root triangle on e: every common neighbour sits in the same Krausz cell as a, b)
else
    exactly one $c \in N$ is non-adjacent to the rest of N — the root-triangle vertex closing a and b at their shared endpoint (Definition 4.10); $C_0 \leftarrow (\{a, b\} \cup N) \setminus \{c\}$, of size $t + 1$, and leave c unassigned to be picked up by the clique at a's or b's other endpoint
Initialize the clique family $\mathcal{C} \leftarrow \{C_0\}$;
while *some edge of H is unassigned* **do**
    Take an unassigned edge incident to a placed vertex and determine the unique clique that must contain it (a maximal mutually-adjacent, consistently-covered set);
    Create a new cell or extend an existing one accordingly;
    if *some vertex would lie in 3 cliques, or some edge in 2, or two cliques meet in > 1 vertex* **then**
        return REJECT;
Build G: one vertex per clique of $\mathcal{C}$; each vertex of H becomes the edge joining its two cliques (pendant if covered once);
return G;

Theorem 5.3 (Roussopoulos [7])**.** *Algorithm 1 decides whether H is a line graph and, if so, outputs a root graph, in $O(n + m)$ time, where $n = |V(H)|$ and $m = |E(H)|$.*

5.3 Lehot's Algorithm

Lehot [8] gave a linear-time algorithm with a different organization. Rather than the Krausz constraints directly, it uses the van Rooij–Wilf local structure (Theorem 4.11): triangles are classified as odd or even (Definition 4.10), and the star cliques covering each vertex are read off from this classification. Crucially, Lehot *constructs a candidate root first and then verifies*: it computes a tentative G, forms $L(G)$, and compares it with the input H; agreement certifies a line graph, disagreement certifies a non-line-graph.

Algorithm 2: Lehot recognition and inversion (outline)

Input: a connected graph H
Output: a root graph G with $L(G) \cong H$, or REJECT
Handle base cases, including the K_3/claw ambiguity, explicitly;
for *each vertex v of H* **do**
  Classify the triangles on v as odd or even (Def. 4.10);
  Partition the closed neighborhood of v into the candidate star cliques covering it;
Assemble the local cliques into a global family $\mathcal{C}$ with each vertex in ≤ 2 cliques (Krausz form);
Build the candidate root G from $\mathcal{C}$ (clique $\mapsto$ vertex; H-vertex $\mapsto$ edge between its two cliques; pendants for once-covered vertices);
Recompute $L(G)$ and test the van Rooij–Wilf conditions on H (no induced $K_{1,3}$; two odd triangles on a shared edge span K_4);
if $L(G) \cong H$ **then**
  return G;
else
  return REJECT;

Theorem 5.4 (Lehot [8])**.** *Algorithm 2 recognizes line graphs and outputs a root graph in $O(n + m)$ time.*

6 Line Graph Modification Problems

Real graphs are rarely *exactly* line graphs: a single induced claw already rules it out (Corollary 4.5). So instead we can ask how far a graph is from being one: the fewest edges you would need to delete, add, or either, to reach a line graph. Each version gives a decision problem: given G and a budget k, is that distance at most k?

Problem 6.1 (Line-Graph Deletion / Completion / Editing)**.** Given G and k, decide whether some combination of at most k edge deletions (only, for the deletion variant), additions (only, for completion), or both (for editing) turns G into a line graph.

Line graphs are defined by a finite list of forbidden induced subgraphs (Beineke, Theorem 4.13), none with more than $\nu = 6$ vertices, and this alone is enough to make all three problems tractable for small k. Call a problem *fixed-parameter tractable* (FPT) if it can be solved in time $f(k) \cdot n^{O(1)}$ for some function f depending only on k: no matter how bad f is, the graph size n only ever enters polynomially. Here the FPT algorithm is simple (Cai [11]): find a forbidden subgraph, try each of its $\binom{6}{2} = 15$ edge slots as the fix, recurse with budget $k - 1$, and give up once the budget is spent. This runs in $O(15^k \cdot n^7)$ time.

All three problems are also NP-complete. Yannakakis [13] proved this directly for line graphs, and it also follows from a general dichotomy of Aravind, Sandeep, and Sivadasan [14]: the claw by itself is already enough to push deletion, completion, and editing into NP-complete territory.

Deletion is the one case where more is known. Eiben and Lochet [15] found a polynomial kernel with $O(k^5)$ vertices, closing a question Hüffner had left open since 2013 (WorKer). Their idea: pack in as many edge-disjoint copies of Beineke's forbidden subgraphs as will fit. Their union forms a witness set S with $O(k)$ vertices; this is the "union of forbidden subgraphs" step. Growing S a little further, using the clique structure of the rest of the graph, gives a reduced graph G' for which solving Line-Graph-Edge Deletion on (G, k) is equivalent to solving it on (G', k). Finding G' only takes polynomial time, since the forbidden subgraphs are so small. What the paper does *not* do is solve the reduced instance: a kernel only shrinks the input, so Line-Graph-Edge Deletion on G' is still exactly as hard as on G.

No such kernel is known for completion or editing: fixing one forbidden subgraph by adding an edge can create a new one elsewhere, so the packing argument breaks down. There is also a concrete reason to doubt one exists: Kratsch and Wahlström found a specific 7-vertex forbidden subgraph for which deletion and editing provably have no polynomial kernel, unless a widely-disbelieved collapse of complexity classes

turns out to be true. So the gap between deletion and the other two may be permanent, not just a hole in current techniques.

Editing and inverting can also be fused into one step: Kandanaarachchi, Kilby, and Ong [21] pose editing to the nearest line graph as an integer program and invert the result, giving a pseudo-inverse for non-line graphs.

7 Line Graphs and Machine Learning

Most graph neural networks (GNNs) pass messages between the *vertices* of G: each layer updates a vertex's representation from those of its neighbours. Many tasks of interest, however, are naturally attached to *edges* or to pairs of vertices: predicting a missing link, deciding whether two edges belong to the same community, scoring an interaction between two entities. Reading off such a quantity from a node-level GNN forces some pooling step (e.g. combining the two endpoint embeddings of a candidate edge), which can lose information. The line graph offers a direct fix: since $V(L(G)) = E(G)$ (Proposition 3.4), an edge-level quantity on G is exactly a vertex-level quantity on $L(G)$, so ordinary vertex-level GNN machinery becomes applicable without any pooling.

7.1 Link Prediction on the Line Graph

Cai, Li, Wang, and Ji [18] recast link prediction as node classification on a line graph. Given a candidate pair of vertices (u, v), they extract an enclosing subgraph around u and v (in the style of SEAL-type subgraph methods) and pass to its line graph: the candidate link itself, together with the edges already present at u and v, become vertices of this line graph, and the link's existence becomes a label on *one specific vertex* rather than a joint property of two vertices. A GNN then classifies that vertex directly, instead of computing two node embeddings and pooling them into an edge score.

7.2 Community Detection via the Line Graph

Chen, Li, and Bruna [19] use the line graph for supervised community detection on graphs drawn from stochastic block models. Their architecture augments a standard GNN with the *non-backtracking operator*: a variant of the line-graph adjacency in which two (directed) edges are linked only if the second does not double back along the first. Message passing over this operator lets the network propagate information about edge-to-edge relationships that a purely vertex-based GNN cannot see directly.

7.3 Combining a Graph and Its Line Graph: ncRNA–Protein Interaction

Han and Zhang [20] give an example where both G and $L(G)$ are used together. Their method, ncRPI-LGAT, predicts interactions between non-coding RNAs and proteins: for a candidate ncRNA–protein pair, it first extracts a local enclosing subgraph of the ncRNA–protein network G around that pair (as in Section 7.1), then converts this subgraph into its line graph so that the pair to be classified again becomes a single vertex. An attention-based GNN (GATv2) is run on the line graph to produce the final prediction. Structure is thus read twice: once when the enclosing subgraph is built in G, and once when message passing runs on $L(G)$ of that subgraph.

8 Conclusion

The line graph is an almost lossless change of perspective: its vertices are the edges of G, and Whitney's theorem (Theorem 4.3) says it determines G up to isomorphism, apart from a single small exception. Several independent characterizations certify exactly which graphs are line graphs, and whenever one is, a linear-time algorithm recovers its unique root: no information is lost in either direction.

That same strength makes it impractical. The map trades n vertices for m, and any vertex of degree ≥ 3 already makes it bigger than G: it inflates, it does not compress. Worse, exact inversion needs an exact line graph, which real graphs almost never are, and repairing one into a line graph first is NP-complete. The machine-learning uses of Section 7 accordingly all go forward, from G to $L(G)$, and never invert it. Whether a cheap approximate inverse could make this lossless view usable in practice is an open question.

References

[1] Euler, L. Solutio problematis ad geometriam situs pertinentis. *Commentarii academiae scientiarum Petropolitanae*, 8:128–140, 1741. (Presented 1736; Eneström index E53.)

[2] Hierholzer, C. Über die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren. *Mathematische Annalen*, 6(1):30–32, 1873.

[3] Whitney, H. Congruent graphs and the connectivity of graphs. *American Journal of Mathematics*, 54(1):150–168, 1932.

[4] Krausz, J. Démonstration nouvelle d'un théorème de Whitney sur les réseaux. *Mat. Fiz. Lapok*, 50:75–85, 1943.

[5] van Rooij, A. C. M. and Wilf, H. S. The interchange graph of a finite graph. *Acta Mathematica Academiae Scientiarum Hungaricae*, 16(3–4):263–269, 1965.

[6] Beineke, L. W. Characterizations of derived graphs. *Journal of Combinatorial Theory*, 9(2):129–135, 1970.

[7] Roussopoulos, N. D. A $\max\{m, n\}$ algorithm for determining the graph H from its line graph G. *Information Processing Letters*, 2(4):108–112, 1973.

[8] Lehot, P. G. H. An optimal algorithm to detect a line graph and output its root graph. *Journal of the ACM*, 21(4):569–575, 1974.

[9] Naor, J. and Novick, M. B. An efficient reconstruction of a graph from its line graph in parallel. *Journal of Algorithms*, 11(1):132–143, 1990.

[10] Degiorgi, D. G. and Simon, K. A dynamic algorithm for line graph recognition. *Graph-Theoretic Concepts in Computer Science (WG'95)*, LNCS 1017, 37–48, 1995.

[11] Cai, L. Fixed-parameter tractability of graph modification problems for hereditary properties. *Information Processing Letters*, 58(4):171–176, 1996.

[12] Lewis, J. M. and Yannakakis, M. The node-deletion problem for hereditary properties is NP-complete. *Journal of Computer and System Sciences*, 20(2):219–230, 1980.

[13] Yannakakis, M. Edge-deletion problems. *SIAM Journal on Computing*, 10(2):297–309, 1981.

[14] Aravind, N. R., Sandeep, R. B., and Sivadasan, N. Dichotomy results on the hardness of H-free edge modification problems. *SIAM Journal on Discrete Mathematics*, 31(1):542–561, 2017.

[15] Eiben, E. and Lochet, W. A polynomial kernel for line graph deletion. *28th Annual European Symposium on Algorithms (ESA)*, LIPIcs 173, 42:1–42:15, 2020. `arXiv:2006.15584`.

[16] Crespelle, C., Drange, P. G., Fomin, F. V., and Golovach, P. A. A survey of parameterized algorithms and the complexity of edge modification. *Computer Science Review*, 48:100556, 2023.

[17] Harary, F. *Graph Theory*. Addison-Wesley, 1969.

[18] Cai, L., Li, J., Wang, J., and Ji, S. Line graph neural networks for link prediction. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2021. `arXiv:2010.10046`.

[19] Chen, Z., Li, X., and Bruna, J. Supervised community detection with line graph neural networks. *International Conference on Learning Representations (ICLR)*, 2019. `arXiv:1705.08415`.

[20] Han, Y. and Zhang, S.-W. ncRPI-LGAT: prediction of ncRNA–protein interactions with line graph attention network framework. *Computational and Structural Biotechnology Journal*, 21:2286–2295, 2023.

[21] Kandanaarachchi, S., Kilby, P., and Ong, C. S. A pseudo-inverse of a line graph. arXiv:2508.09412, 2025.