Linnet

Half-edge graphs and subgraph-first algorithms

1 Overview

Linnet is a graph library for tensor networks, Feynman diagrams, and other workflows where the boundary of a subgraph matters. Its central representation is a half-edge graph. An ordinary internal edge is a pair of half-edges, while an external edge has one dangling half-edge. A node owns incident half-edges, and a subgraph is a selection of half-edges rather than a detached copy of the graph.

> **Why half-edges**
> Cutting an internal edge can turn its two halves into boundary edges without changing the degree of either endpoint. This makes operations such as extraction, excision, contraction, sewing, traversal, and cut analysis natural for physics graphs. Linnet algorithms therefore accept subgraph views even when the caller wants to operate on the full graph.

1.1 Choose a task

- To add Linnet to a Rust project and validate a graph with a boundary, follow the Rust guide and the exact `HedgeGraphBuilder` reference.
- To parse and inspect a DOT graph from Python, follow the source-built Python guide.
- To select an internal subgraph and compute a cycle basis, continue with the first-graph tutorial.
- To enumerate cycles, cuts, or connected regions, use the graph-algorithms guide with the `HedgeGraph` reference.
- To render a tree of existing DOT files and rebuild only changed figures, use the Clinnet command-line guide.
- To draw an editable graph in Typst, begin with the Typst guide, then continue to the Linnest Typst guide layout coordinates are deliberately separate from graph identity.

1.2 A working model

A Linnet workflow has four parts:

- construct a graph with `HedgeGraphBuilder`, or parse a DOT graph;
- retain the typed identifiers returned for half-edges, nodes, and edges;
- describe the region of interest with one of the subgraph representations;
- run traversal or mutation operations against that subgraph and the graph's involution.

The involution is the pairing map for half-edges. A paired entry represents an internal edge; an identity entry represents a dangling edge and records its flow. Edge orientation and flow are related concepts, but they are not interchangeable. Consult the API reference for the exact variants and conversion rules in your Linnet release.

> **Indexes belong to one graph**
> `Hedge`, `NodeIndex`, and `EdgeIndex` are compact typed indexes, not durable object IDs. Graph edits can change storage and mappings. Keep the result returned by an operation, and do not apply an index or subgraph bitset to an unrelated graph just because its numeric values happen to fit.

1.3 Construction and interchange

The Rust package can be added directly from crates.io:

```toml
[dependencies]
linnet = "0.17"
```

Programmatic construction uses `HedgeGraphBuilder`. DOT support provides a convenient human-readable interchange and debugging format; it can carry global, node, edge, and half-edge attributes. Treat DOT as a representation boundary, not as a substitute for the typed graph invariants. Validate or handle parser errors before running algorithms.

The `drawing` feature adds layout support. Layout is separate from graph identity: coordinates may change without changing the topology. Linnet itself does not provide a supported renderer; use DOT, Clinnet, or Linnest at the presentation boundary.

1.4 Related crates

One graph layer, several consumers
Spenso uses Linnet for tensor-network connectivity and adds tensor contraction semantics. GammaLoop uses the graph model in collider workflows and adds process generation, integration, and persistent state.

The rendered guides, native Rustdoc, and generated Python API are the canonical references. Contributors can follow the stores, invariants, mutation flow, and feature boundaries in the Linnet implementation architecture.

2 Build your first Linnet graph

Linnet's half-edge graph model is available through three surfaces. Choose the language that owns the graph in your workflow; the same concepts—paired internal half-edges, explicit boundaries, typed identifiers, and validated topology—carry across them.

Rust
Use the published `linnet` crate when a Rust application owns graph construction, algorithms, or serialization. This is the most complete and thoroughly tested interface.

Use Linnet from Rust →

Python
Use `linnet_py` for DOT-backed graph inspection and algorithms from Python. The binding is currently a source-built developer preview rather than a published wheel.

Use Linnet from Python →

Typst
Use Linnest when a Typst document owns graph construction, layout, and the final CeTZ drawing. The package is currently bundled with the repository rather than Typst Universe.

Use Linnet from Typst →

Begin with Rust for application code, Python for an existing Python graph workflow, or Typst when the desired result is an editable figure. The interface guide explains where layout, rendering, and interchange belong.

3 Using Linnet from Rust

Use Linnet when a problem is naturally expressed as a half-edge graph with an explicit boundary. This example builds the smallest such graph, validates its involution, and emits DOT that can be inspected or handed to a renderer.

3.1 Create a Rust project

```
cargo new linnet-quickstart
cd linnet-quickstart
cargo add linnet@0.17
```

Replace `src/main.rs` with:

```
use linnet::half_edge::{
    builder::HedgeGraphBuilder, involution::Flow, HedgeGraph,
};

fn main() {
    let mut builder = HedgeGraphBuilder::new();
    let source = builder.add_node("source");
    let sink = builder.add_node("sink");

    builder.add_edge(source, sink, "propagator", false);
    builder.add_external_edge(source, "incoming", false, Flow::Sink);
    builder.add_external_edge(sink, "outgoing", false, Flow::Source);

    let graph: HedgeGraph<&str, &str> = builder.build();
    graph.check().expect("the half-edge involution is valid");

    assert_eq!(graph.n_externals(), 2);
    println!("{}", graph.dot_label(&graph.full_filter()));
}
```

Run it with `cargo run`. The output is a DOT graph with one paired internal edge and two dangling boundary edges. `graph.check()` is the important first invariant: run it immediately after constructing or parsing a graph, before applying graph algorithms.

> **Half-edges make the boundary explicit**
> An internal edge is a pair under the graph's involution. A dangling external edge maps to itself and carries a `Flow`. The `NodeIndex` values returned by the builder belong to this graph; do not reuse their numeric representation as a durable identifier.

Continue with the graph tutorial to select a subgraph and compute a cycle basis. Use the Clinnet guide to render existing DOT files, or the Typst guide to construct and draw an editable graph with Linnest.

`linnet` is a library, so add it to a project with `cargo add`; `cargo install` and Cargo Binstall apply to executables. The separately versioned Clinnet renderer can be installed with `cargo install clinnet --locked`.

4 Using Linnet from Python

The `linnet_py` extension provides DOT-backed Linnet graphs to Python 3.10 and newer. It is a real binding with runtime tests, but it is not currently published to PyPI.

> **Developer preview: build from source**
> There is no supported `pip install linnet-py` release yet. The similarly named `linnet` project on PyPI is unrelated. Use this page from a GammaLoop checkout until official `linnet-py` wheels are published.

4.1 Build the extension

Install a stable Rust toolchain and Python 3.10 or newer, then run:

```
git clone https://github.com/alphal00p/gammaloop.git
cd gammaloop
python3.10 -m venv .venv
. .venv/bin/activate
python -m pip install "maturin>=1.7,<2.0"
maturin develop --locked \
  --manifest-path crates/linnet-py/Cargo.toml \
  --features extension-module,abi3-py310
```

4.2 Parse and inspect a graph

Save this as `linnet_quickstart.py`:

```
import linnet_py as lp

graph = lp.DotGraph.from_string(
    """
    digraph G {
      A;
      B;
      A -> B;
    }
    """
)

whole = graph.full_filter()
assert graph.n_nodes() == 2
assert graph.n_edges() == 1
assert len(graph.iter_nodes_of(whole)) == 2
assert len(graph.iter_edges_of(whole)) == 1

print(graph.dot())
```

Run `python linnet_quickstart.py`. Success means the assertions pass and the graph is printed as DOT. `DotGraph` also exposes typed nodes, half-edges, subgraphs, cycles, oriented cuts, and traversal trees; use the Python reference for the exact current surface.

Use the Rust guide when you need Linnet's complete crate API, or the Typst guide when the final result should be a drawing.

5 Using Linnet from Typst

Linnest brings Linnet's graph model, layout algorithms, and physics-aware drawing styles into Typst through a bundled WebAssembly plugin. This first document draws two nodes joined by one internal edge.

> **Bundled source package**
> Linnest `0.1.0` is not currently published on Typst Universe. Keep the repository's complete `crates/linnest/typst/` directory together: its Typst sources load the adjacent `linnest.wasm` plugin. Typst `0.15.0` or newer is required.

5.1 Create the document

From a GammaLoop checkout, save this as `linnest-quickstart.typ` in the repository root:

```
#import "crates/linnest/typst/src/lib.typ": draw, graph, layout

#let g = graph.build({
  graph.node(<left>, label: [left])
  graph.node(<right>, label: [right])
  graph.edge(
    graph.source(<left>),
    <propagator>,
    graph.sink(<right>),
    label: [$p$],
  )
})

#draw(layout(g))
```

Compile it from the same repository root:

```
typst compile --root . linnest-quickstart.typ linnest-quickstart.pdf
```

The PDF should contain two labeled nodes and one labeled edge. Construction, layout, and drawing are separate stages: keep the graph value when you need queries or subgraphs before rendering it.

Continue with the Linnest guide for DOT parsing, explicit styling, and subgraph drawing, then use the Typst API reference for graph, layout, drawing, physics, and subgraph functions.

6 Tutorial

In this tutorial, you will construct a half-edge graph in Rust, validate its invariants, find its one independent cycle inside an explicitly selected subgraph, and print DOT. The builder returns typed node indexes; Linnet's subgraph filters then select the half-edges on which an algorithm operates.

For the shortest installation and first-run path, begin with Using Linnet from Rust. Then return here to work with a boundary, subgraphs, and cycle algorithms.

6.1 Prerequisites

Create a Rust binary project and add the current Linnet release:

```
cargo new linnet-first-graph
cd linnet-first-graph
cargo add linnet@0.17
```

No drawing feature or external layout program is needed to construct the graph and emit DOT.

6.2 Build a graph with a boundary

Replace `src/main.rs` with the following program:

```
use linnet::half_edge::{
    builder::HedgeGraphBuilder,
    involution::Flow,
    subgraph::{SubSetOps, SuBitGraph},
    HedgeGraph,
};

fn main() {
    let mut builder = HedgeGraphBuilder::new();
```

```
    let a = builder.add_node("a");
    let b = builder.add_node("b");
    let c = builder.add_node("c");

    builder.add_edge(a, b, "ab", false);
    builder.add_edge(b, c, "bc", false);
    builder.add_edge(c, a, "ca", false);
    builder.add_external_edge(a, "incoming", false, Flow::Sink);
    builder.add_external_edge(c, "outgoing", false, Flow::Source);

    let graph: HedgeGraph<&str, &str> = builder.build();
    graph.check().expect("the half-edge involution is valid");

    let boundary: SuBitGraph = graph.external_filter();
    let internal = graph.full_filter().subtract(&boundary);
    let (basis, _) = graph.cycle_basis_of(&internal);
    assert_eq!(basis.len(), 1);
    println!("{}", graph.base_dot());
}
```

Run it with `cargo run`. Success means the assertion passes and stdout contains a DOT `digraph` with nodes `a`, `b`, and `c`, three paired internal edges, and two dangling boundary edges. The external edges do not add a cycle, so the triangle's cycle-basis rank remains one.

> **Verification scope and cost**
> The docs harness compiles and runs this Rust program; it syntax-checks the setup commands without creating a project or using the network. Success requires `graph.check()` and the subgraph cycle-rank assertion to pass. The run is small; a clean external Cargo build is dependency-dominated and can take minutes, while the graph operation itself should finish in well under a second.

> **Keep the returned indexes with their graph**
> `a`, `b`, and `c` are compact `NodeIndex` values owned by this graph construction. Do not serialize their numeric representation as a durable identity or reuse them against another graph. Mutating operations return the mappings needed to relate old and new storage.

6.3 Make the first useful variation

Add a fourth internal edge between `a` and `b`, run the program again, and change the assertion to `basis.len() == 2`. This demonstrates the key distinction: parallel paired half-edges add an independent cycle, while dangling identity half-edges describe the graph boundary excluded by `internal`.

From here, use a subgraph view to run traversal, cuts, or contraction against only the region you intend to modify. Enable Linnet's `drawing` feature only when you need layout rather than plain DOT interchange.

6.4 Troubleshooting and next steps

- A type-inference error around `build()` means Rust cannot choose node storage or data types; keep the explicit `HedgeGraph<&str, &str>` annotation from the example.
- A failed `check()` points to an involution or storage invariant. Validate immediately after parsing or construction, before applying graph algorithms.
- If an algorithm includes an unexpected dangling edge, check that you passed `internal` rather than `graph.full_filter()`, then inspect the `Flow` assigned to each external edge.
- Continue with the subgraph-and-algorithms manual, then consult the Rust API reference for the exact return mappings of mutating operations and additional builder methods.

7 Subgraphs, traversal, cuts, and drawing

Linnet algorithms operate on a graph together with an explicit subgraph view. This lets one algorithm work on the full graph, a selected region, or the boundary created by cutting paired half-edges. Choose the subgraph representation by operation: bit-backed sets are efficient for repeated set algebra, while borrowed or mapped views avoid copying when identity must remain attached to the owning graph.

7.1 Traversal and connectivity

A traversal needs a starting half-edge or node, the active subgraph, and a policy describing which adjacent half-edges may be crossed. Results retain typed indexes into the original graph. Breadth-first traversal is useful for distance layers and connectivity; depth-first traversal is useful for trees, back edges, and decompositions. A traversal tree is a derived view, not a new graph, so mutations must be coordinated with the indexes it contains.

Connectivity and traversal results depend on the chosen subgraph. Run an operation on the full graph only when external half-edges and excluded nodes should participate in the answer. Linnet does not currently expose a dedicated articulation-point or biconnected-component API; derive those from traversal data only when your application owns that algorithm and its tests.

7.2 Cycles and cuts

`cycle_basis` returns an independent cycle vector together with the spanning forest used to construct it. Use `cyclotomatic_number` when the numerical cycle rank itself is the result. `all_cycles` expands a basis and can grow exponentially; prefer a basis when assigning loop momenta. Symmetric differences combine cycle bitsets without inventing new graph identities.

Cut enumeration separates required left and right node sets and returns the left region, cut content, and right region. A cut edge is represented through its half-edges, preserving which side owns each endpoint. Validate direction/flow after enumeration when a physical Cutkosky or tensor-network interpretation depends on orientation.

> **Complexity is part of the API choice**
> Enumerating every cycle or every admissible cut is inherently combinatorial. Filter the subgraph and endpoint sets first, and use basis or connectivity operations when enumeration is not required by the caller.

7.3 Enumerate separating cuts

This complete example builds a square with a diagonal and asks for every cut separating opposite nodes. It is the smallest graph that makes the distinction between a path and a cut family visible without external data or drawing support.

```
use linnet::half_edge::{HedgeGraph, builder::HedgeGraphBuilder};

fn main() {
    let mut builder = HedgeGraphBuilder::new();
    let a = builder.add_node(());
    let b = builder.add_node(());
    let c = builder.add_node(());
    let d = builder.add_node(());

    builder.add_edge(a, b, (), true);
    builder.add_edge(b, c, (), true);
    builder.add_edge(c, d, (), true);
    builder.add_edge(d, a, (), true);
    builder.add_edge(b, d, (), true);

    let graph: HedgeGraph<(), (), ()> = builder.build();
```

```
    graph.check().expect("the half-edge involution is valid");
    let cuts = graph.all_cuts_from_ids(&[a], &[c]);

    assert_eq!(cuts.len(), 4);
    println!("separating cuts: {}", cuts.len());
}
```

The expected invariant is four distinct left/cut/right partitions separating `a` from `c`. Changing the diagonal changes that count, while changing only node or edge payloads does not. The native `HedgeGraph` Rustdoc and builder Rustdoc give the exact compiled type boundaries for this revision.

> **Interpret a cut mismatch structurally**
> A failed `graph.check()` indicates a storage or involution invariant and must be resolved before enumeration. Zero cuts usually means the endpoint sets overlap or are disconnected from the selected region. An unexpected nonzero count means the constructed edges or endpoint sets do not describe the graph you intended; inspect those before adding post-enumeration filters.

7.4 Mutation boundaries

Extraction copies a selected region, excision separates a graph along its boundary, sewing joins compatible dangling half-edges, and contraction identifies structure. These operations return mappings or replacement indexes where needed. Keep those results: a numeric `Hedge` or `NodeIndex` from before a storage-changing operation is not automatically a valid index into the resulting graph.

7.5 DOT and drawing

DOT parsing and emission are interchange/debugging boundaries. Attributes can carry graph, node, edge, and half-edge data, but the typed Rust invariants are reconstructed by the parser; handle parser errors before running algorithms. The `drawing` feature enables layout; rendering belongs to a DOT consumer, Clinnet, or Linnest. Layout coordinates are presentation data and may vary without changing topology, subgraph membership, or edge flow.

> **Tensor contraction uses Spenso**
> Linnet determines connectivity and graph transformations. Tensor compatibility, contraction cost, and execution belong to Spenso even when a Spenso network delegates its graph algorithms to Linnet.

8 Render DOT figures with Clinnet

Clinnet is Linnet's command-line companion for documentation and diagram workflows. The Cargo package is named `clinnet`, while the installed executable is named `linnet`. It recursively finds DOT files, renders one PDF per graph through Typst, and assembles those PDFs into a grid. Use the Rust library for graph algorithms; use Clinnet when the input is already DOT and the desired result is a browsable set of figures.

8.1 Install and render a directory

Clinnet requires Typst 0.15.0 or newer. Install both commands, then confirm the executable's current flags rather than relying on an older pre-subcommand invocation:

```
cargo install clinnet
cargo install typst-cli --version 0.15.0 --locked
linnet --help
linnet draw examples
```

The final command scans `examples/` recursively. With default paths it creates:

- `build/figs/`, mirroring the input directory and containing one PDF per DOT file;
- `build/grid.pdf`, the combined document;
- `build/templates/`, containing the default figure, grid, layout, Linnest, and Kurvst assets;
- `build/.cache/figures.json`, the per-figure content cache; and
- `build/.cache/run-metadata.json`, the plan used by targeted redraw commands.

> **What a second run reuses**
> A figure is reused only when its DOT source, selected figure template, explicit `--style` files, `--input` values, and the bundled `.typ`/`.wasm` assets in `build/templates/` have the same content hash. Clinnet does not recursively discover arbitrary imports beside a custom template; pass those files with `--style` when they must invalidate the cache. The grid is still regenerated from the current figure index. Removing a DOT file removes its stale figure output during the next full `draw`; deleting the figure cache forces every remaining figure to rebuild.

8.2 Customize templates and inputs

Use `--figure-template` for each individual figure and `--grid-template` for the combined PDF. Repeat `--style` for extra files whose contents must invalidate the figure cache. Repeat `--input key=value` to expose strings through Typst's `sys.inputs` in both compilation stages:

```
linnet --build-dir build draw graphs \
  --figure-template templates/figure.typ \
  --grid-template templates/grid.typ \
  --style templates/colors.typ \
  --input theme=dark \
  --input scale=1.5 \
  --output-path build/review.pdf
```

Clinnet supplies `data-path`, `data_path`, and a derived `title` to the figure template in addition to caller inputs. `--output-path` selects the final grid PDF; it is not a figure-template input. Input values are strings; parse them in Typst when a number or boolean is required. The embedded defaults include the canonical Linnest and Kurvst package trees, so a custom template can use the same graph-layout and curve primitives. Continue to the Linnest Typst API for graph construction, layout, inspection, and drawing rather than copying its wrapper surface into a Clinnet template guide.

> **Templates must share a Typst project root**
> Clinnet chooses the narrowest existing directory that contains the template and its inputs. A Typst import outside that tree, a missing style file, or a template moved without its dependencies fails before a PDF is produced. Keep the templates and imported assets beneath one project directory and read the full Typst error printed by Clinnet.

8.3 Make a targeted rebuild

Run a complete `draw` once so that the build metadata records the input root, templates, output paths, styles, and figures. You can then rebuild one known DOT figure without rescanning the directory, or rebuild only the grid:

```
linnet --build-dir build redraw-figure graphs/box.dot --input theme=light
linnet --build-dir build redraw-grid --output-path build/review.pdf
```

`redraw-figure` starts with the inputs recorded by the full draw and applies new inputs as per-key overrides. `redraw-grid` reuses the cached figure list; if any `--input` is supplied to that command, the supplied list replaces the grid's recorded input list, so repeat every value the grid template still needs. A target not present in the preceding plan must be added with a new full `draw`.

Use `--no-parallel` for deterministic sequential diagnosis, or `--jobs N` to bound parallel figure compilation. These options affect scheduling, not cache identity or graph semantics.

8.4 Source and release boundaries

The implementation and exact Clap definitions are in the Clinnet command source. Clinnet has its own package version and version history, independent of the Rust `linnet` library. Record the Clinnet and Typst versions when reporting a rendering or cache problem.

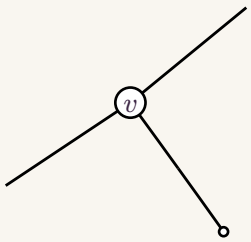
9 Build and draw with Linnest

Linnest is the Typst and WebAssembly interface to Linnet's graph model, DOT parser, layout algorithms, graph queries, and physics-aware drawing styles. This guide follows one graph from construction through layout and drawing; the separate Typst API reference carries the supported symbol inventory.

Choose the interface deliberately
Use Linnet's Rust or Python interfaces for graph construction and algorithms in an application. Use Linnest when the graph, layout, or final drawing is owned by a Typst document.

9.1 Linnet

The `linnet` crate wrapped by this package is built around a half-edge graph data structure. This means that instead of a graph being represented as a set of nodes and edges, it is represented as a set of half-edges H, and a set of vertices V. The graph structure is then encoded through two maps. The first map, $\partial : H \longrightarrow V$ maps each half-edge to its corresponding vertex. The preimage of any vertex v is the set of half-edges that map to it, called the crown of v. The second map, $\iota : H \longrightarrow H$, is an involution that *glues* half-edges together to form edges. If a half-edge is glued to itself, we call that an external half-edge. This means that linnet graphs are strictly more capable than normal edge and vertex graphs.



Native half-edges also make subgraphs more granular because they can be encoded as sets of half-edges. This directly supports vertex-induced subgraphs: the union of the crowns of a set of vertices.

9.2 Linnest

Linnest is the Typst and WebAssembly interface to Linnet. It provides layout algorithms, DOT parsing, and selected graph algorithms without a separate runtime process.

Graphs can be constructed in two ways: parse a DOT string with `parse`:

```
#let g = parse("digraph { a -> b }")
```

or build from edges and nodes with `build`, using a Fletcher-inspired syntax:

```
#let g = build({
  node(<a>, label: [$v$])
  node(<b>)
  edge(source(<a>), <a-b>, sink(<b>), label: [e])
```

```
  edge(source(<a>), <in-a1>, label: [e])
  edge(source(<a>), <in-a2>, label: [e])
})
```

In either case, `type(g)` is `dictionary`: graph values wrap an archived Linnet graph together with native Typst data. Rust owns topology, layout state, statement metadata, and internal opaque payload bytes. User data captured from Typst stays in Typst and is merged back into query records.

The main use case is to place nodes and edges on a canvas with the `layout` function and render the result with `draw`.

- `graph` for construction, parsing, inspection, joins, and graph algorithms.
- `subgraph` for subgraph object construction and inspection.
- `layout` for the separate layout pass.
- `draw` for rendering a laid-out graph object with CeTZ.
- `physics` for reusable particle-line edge styles.

9.2.1 Choose an import path

Linnest and Kurvst are currently bundled source packages, not Typst Universe packages. A Clinnet run writes both package trees below `build/templates/`. From a custom template in that directory, import Linnest with:

```
#import "crates/linnest/typst/src/lib.typ": draw, graph, layout, subgraph
```

From this repository's `crates/linnest/typst/examples/` directory, the equivalent checkout-relative import is `../src/lib.typ`. Keep the package directory and its `linnest.wasm` file together when copying it elsewhere. The examples below use the checkout-relative form because they are also compiled as repository tests.

9.2.2 Minimal Build Example

```
#import "../src/lib.typ": draw, graph, layout, subgraph
#import graph: build, dot, edge, edges, node, nodes, parse, sink, source

#let g = build({
  node(<a>)
  node(<c>)
  edge(
    source(<a>, compass: "e"),
    <a-c>,
    sink(<c>, compass: "w"),
    label: [a-c],
    statements: (
      color: "0055ff",
      source-color: "d72638",
      sink-color: "1b7f4c",
    ),
  )
},
  name: "demo",
)
#let g = layout(g)
#let east = subgraph.compass(g, "e")
#let edge-records = edges(g, subgraph: east)
#let dot-text = dot(g)
#let edge-label(edge) = text(fill: rgb("#" + edge.color))[#edge.label]
#let source-style(edge) = (stroke: rgb("#" + edge.source-color) + 0.5pt)
#let sink-style(edge) = (stroke: rgb("#" + edge.sink-color) + 0.5pt)
#context if target() == "paged" {
  draw(
```

```
    g,
    subgraph: east,
    edge-label: edge-label,
    edge-label-style: (anchor: "south"),
    source-style: source-style,
    sink-style: sink-style,
  )
} else {
  [The downloadable PDF renders this CeTZ result. The HTML manual keeps the
  copyable source because Typst's experimental HTML target does not yet emit
  the drawing content.]
}
```

9.2.3 Graph Objects

Graph values combine archived Linnet topology with native Typst data. The focused graph reference documents their constructors, transforms, queries, and update operations.

9.3 Continue into the API

Use the graph module for construction and queries, the layout functions for placement, and the drawing functions for CeTZ output. Physics styles and subgraph objects have their own focused reference pages.

10 Choose a Linnet interface

Use Rust when Linnet is part of an application or library, Python for standalone DOT-backed graph workflows, and Linnest when a Typst document owns layout and drawing. Each interface shares the half-edge model while exposing the conventions natural to its language.

10.1 Typst package

Linnest combines Typst functions with a WebAssembly plugin. Follow the Linnest workflow guide for a first drawing, then use the focused Typst API reference for its graph, layout, drawing, physics, and subgraph surfaces. The curve helpers are Kurvst re-exports and stay documented in their canonical GammaLoop guide.

10.2 Rust package

The `linnet` package is organized into these principal areas:

- `half_edge` provides `HedgeGraph`, builders, the involution and edge data, subgraph types, traversal trees, and graph algorithms;
- `parser` provides DOT parsing and DOT-backed graph data;
- `half_edge::layout` provides layout helpers when the `drawing` feature is enabled;
- `permutation`, `tree`, and `union_find` provide supporting algorithms and stores.

The generic graph types separate edge, vertex, and half-edge payloads from the node-storage implementation. Start with the Rust orientation, then use the revision-specific Rustdoc for complete generic bounds, implementations, and method signatures; choose a concrete storage type according to the workflow and mutation behavior you need.

Cargo features enable optional capabilities:

- `serde` adds serialization traits;
- `bincode` adds binary encoding support;
- `drawing` enables Linnet's layout module and its optional geometry dependency;
- `symbolica` adds Symbolica interoperability.

An item shown in an all-features API reference may not be available in a default build. Check its required feature and your `Cargo.toml` before using it.

Layout is not rendering
Linnet can compute layout coordinates, but it does not own a supported Rust renderer. Emit DOT for interchange, use Clinnet for command-line figure batches, or use Linnest when a Typst document owns the final drawing.

10.3 Standalone Python distribution

Distribution and import have different names
Install the Python distribution `linnet-py`, then import `linnet_py`. It requires Python 3.10 or newer. It is a standalone extension and is not a `symbolica.community` module. Its package version is independent of the Rust `linnet` version. Record both versions when diagnosing compatibility between Rust and Python code.

The Python binding focuses on DOT-backed graphs and mirrors typed identifiers and subgraphs:

```python
from linnet_py import DotGraphBuilder

builder = DotGraphBuilder()
left = builder.add_node()
right = builder.add_node()
builder.add_edge(left, right)
graph = builder.build()

print(graph.dot())
```

`DotGraph` can also parse one graph from a string or file, or a set of graphs from a string. The exported classes include `Hedge`, `NodeIndex`, `EdgeIndex`, `Flow`, `Orientation`, graph attribute wrappers, `HedgePair`, `Subgraph`, `Cycle`, `OrientedCut`, `TraversalTree`, `DotGraph`, and `DotGraphBuilder`.

Mutable statement wrappers expose mapping-like access to DOT attributes. Consult the structured Python API for their lifetime and write-through behavior, and do not assume that a copied mapping remains attached to its graph.

11 Linnest Typst API

Linnest documents a small supported surface around Linnet's graph model. Start with the workflow guide if you want to build and draw a graph; use these pages when you need an exact function, parameter, or default.

11.1 Supported surface

- `graph` is a module for construction, parsing, inspection, data transforms, joins, cycles, and forests.
- `layout` and `layout-sequence` are top-level functions. They are documented together because both come from `layout.typ`.
- `draw`, `edge-halves`, and `to-cetz-edge-halves` are top-level drawing functions from `draw.typ`.
- `physics` is a module of reusable particle-line styles and callbacks.
- `subgraph` is a module for constructing and inspecting zero-copy subgraph values.
- `curve` is a re-export of Kurvst, not a second Linnest API. Use the canonical Kurvst curve and path reference.

Modules and functions import differently
Import `graph`, `physics`, and `subgraph` as modules when you want qualified calls such as `graph.build`. Import `layout`, `layout-sequence`, and `draw` directly from `lib.typ`; they are functions rather than nested modules.

```
#import "crates/linnest/typst/src/lib.typ": draw, graph, layout, physics, subgraph
#import graph: build, edge, node, sink, source
```

The same package also exports the `layouts` module for callers that need the underlying layout namespace; its `layouts.layout` and `layouts.sequence` aliases are linked from the layout page. Bindings exposed only as a consequence of Typst module loading, such as serialization helpers or imported dependency modules, remain implementation details rather than supported API. The focused reference uses the supported functions and modules above.

12 graph module

Use `graph` for graph construction, DOT parsing, inspection, data transforms, joins, and graph algorithms. Import the module for qualified calls, or import selected functions from it when a short construction DSL is clearer.

12.1 Concepts

12.1.1 Graph Specs

The Typst construction API is half-edge first: create node items, create source and sink half-edge endpoints, then pass edge items to `graph.build`. `graph.build` accepts both comma-separated items and ordinary Typst code blocks. Typst labels such as `<a>`, `<h1>`, and `<e1>` are API names; node and edge names are emitted back from `nodes(g)` and `edges(g)` as Typst labels. They are resolved before the wire format is sent to the Rust plugin. Numeric `id` arguments choose graph indexes or ordering: on nodes, `id` fixes the resulting node index and must be unique and in bounds. On nodes, edges, sources, and sinks, extra named arguments are captured as opaque Typst data fields: `edge(source(<a>, style: physics.source-stroke()), sink(<b>), particle: "g")` stores `(style: ..)` in the source data and `(particle: "g")` in the edge data. These user data values are not sent through the Rust plugin boundary. Typst sends an internal opaque payload with correlation data, asks Rust to resolve the graph indexes, then stores user data in arrays at those resolved graph, node, edge, and half-edge ids. `graph.info`, `graph.nodes`, and `graph.edges` merge those native values into the returned records. A captured `label` data field on nodes and edges is display content used by the default drawing style; use `statements: (label: "...")` when a flat metadata label string is needed. Statements are flat metadata used by DOT; they cannot nest. Values are scalar strings/numbers/booleans. Use data fields for structured Typst data or content.

`graph.build` and `graph.parse` also accept `default-node-data`, `default-edge-data`, `default-source-data`, and `default-sink-data`. These defaults are merged into the corresponding data; captured data fields on nodes, edges, sources, and sinks override the defaults. For drawing, the physics helpers read `data.style` on source and sink half-edges and `data.display-label`/`data.label` on edges:

```
#let g = build({
  node(<a>, label: [a])
  node(<b>, label: [b])
  edge(
    source(<a>, style: physics.source-stroke(c: red)),
    <e>,
    sink(<b>, style: physics.sink-stroke(c: blue)),
    label: [$p$],
    particle: "g",
  )
},
```

```
  default-edge-data: (kind: "propagator"),
)
#let callbacks = physics.style()
#draw(
  layout(g),
  source-style: callbacks.source-style,
  sink-style: callbacks.sink-style,
  edge-label: callbacks.edge-label,
)
```

When parsing DOT, the same native data arrays can be filled from selected string fields. Rust parses DOT into topology and statement metadata; Typst applies defaults and evaluates selected fields afterward. The selected fields are evaluated with the record's merged `fields` dictionary in scope. Since node names are labels, use `#str(name)` when a name should become visible text:

```
#let g = parse(
  "digraph g { a [label=\"A\"]; a -> b [label=\"$p$\", source=\"out\", sink=\"in\"] }",
  eval-node-fields: ("label",),
  eval-edge-fields: ("label",),
  eval-source-fields: ("statement",),
  eval-sink-fields: ("statement",),
).first()
#nodes(g).first().data.label
#edges(g).first().data.label
```

The same transform can run after construction. This is useful for global edge statements that should apply to every edge while still seeing local edge fields:

```
#let g = build({
  node(<a>)
  node(<b>)
  edge(source(<a>), sink(<b>), statements: (mom: "p"))
}, default-edge-statements: (display-label: "$#mom$"))
#let g = graph.eval-fields(g, eval-edge-fields: ("display-label",))
#edges(g).first().data.at("display-label")
```

```
#let g = build({
  node(<a>)
  node(<c>)
  edge(source(<a>), <e1>, sink(<c>))
})
```

Named nodes and edges can be updated after construction without scanning in the caller. The update replaces the native data, or it can be a callback receiving `(data, record)`:

```
#let g = build({
  node(<a>)
  node(<c>)
  edge(source(<a>), <e1>, sink(<c>))
})
#let g = update-node-data(g, <a>, (label: [A]))
#let g = update-edge-data(g, <e1>, (data, edge) => (
  label: [$p$],
  source: edge.source.node,
))
#node-data(g, <a>).label
#edge-data(g, <e1>).label
```

`source(..)` and `sink(..)` accept a node reference plus optional `name`, `id`, `statement`, and `compass`. `name` is a Typst label name for the half-edge; `id` is numeric:

```
edge(source(<a>, name: <h1>, id: 0), <e1>, sink(<c>, id: 2), label: [a-c])
```

One half-edge creates an external edge. The side is determined by the constructor, so there is no public `flow` argument:

```
edge(<incoming>, sink(<a>))
edge(source(<c>), <outgoing>)
```

`graph.build` does not interpolate statement strings on the Rust side. Use `graph.eval-fields` or `graph.map` when a default statement should turn into Typst content or structured data. The evaluation scope includes the record's merged fields, so default edge statements can still refer to local edge fields:

```
#let g = build({
  node(<a>)
  node(<c>)
  edge(
    source(<a>),
    <a-c>,
    sink(<c>),
    label: [a-c],
    statements: (color: "0055ff", label: "a-c"),
  )
},
  default-edge-statements: (
    color: "000000",
    display-label: "$#label$",
  ),
)
#let g = graph.eval-fields(g, eval-edge-fields: ("display-label",))
```

12.1.2 Graph object API

Graph objects are Typst dictionaries wrapping archived Rust graph bytes plus native Typst data arrays for graph, node, edge, source, and sink data. The Rust graph may also carry an internal opaque payload, but that payload is used only by the Typst wrapper and is not exposed in public records. Build or parse graph objects with `graph`, transform graph objects with `layout`, and pass objects back to `graph` or `subgraph` for inspection. Subgraph objects are still opaque zero-copy values.

- `graph.parse(input)` parses one or more DOT digraphs and returns an array of graph objects. Its `eval-graph-fields`, `eval-node-fields`, `eval-edge-fields`, `eval-source-fields`, and `eval-sink-fields` arguments are convenience arguments for `graph.eval-fields`.
- `graph.build(..)` constructs one graph object from a stream of node and edge items.
- `graph.map(graph, ..)` maps graph, node, edge, source, and sink records to new native data without changing topology.
- `graph.node-data(graph, <name>)` and `graph.edge-data(graph, <name>)` return one named node or edge data.
- `graph.update-node-data(graph, <name>, data)` and `graph.update-edge-data(graph, <name>, data)` update one named node or edge data. Direct replacements and callbacks run in Typst with `(data, record)`.
- `graph.eval-fields(graph, ..)` evaluates selected record fields into data entries. It works on parsed and built graph objects.
- `node(..)` returns a node item.
- `source(..)` and `sink(..)` return half-edge endpoints.
- `edge(..)` returns an edge item built from source/sink endpoints and an optional edge name.

- `layout(graph, ..)` runs layout as an explicit second step. Its settings are named parameters so calls stay descriptive and Tidy can document each field.
- `draw(graph, ..)` draws a laid-out graph object with CeTZ.
- `dot(graph)` returns a DOT string for inspection or export.

12.1.3 Graph Queries

`graph.info(g)` returns graph metadata. `nodes(g)` returns node records, and `edges(g)` returns edge records. Node and edge record `name` values are Typst labels when present. Pass `subgraph: sg` to filter nodes or edges by a subgraph object.

`graph.join(left, right, key: "statement")` joins matching dangling half edges. The key is read from half-edge statements or numeric ids and can be `"statement"`, `"compass"`, or `"id"`.

`graph.cycles(g)` returns subgraph objects for a cycle basis. `graph.forests(g)` returns subgraph objects for spanning forests.

12.2 graph

- build()
- parse()
- node()
- source()
- sink()
- edge()
- group()
- pin()
- start()
- pos()
- map()
- style()
- eval-fields()
- info()
- dot()
- nodes()
- edges()
- node-data()
- edge-data()
- update-node-data()
- update-edge-data()
- join()
- cycles()
- forests()

12.2.1 build

Build one graph object from a stream of node and edge items.

Use `node`, `source`, `sink`, and `edge` to create graph items. Positional items may be passed as comma-separated arguments or yielded from a Typst code block.

12.2.1.1 Parameters

```
build(
  ..items: array,
  name: none string,
  data: any,
  statements: dictionary,
  default-node-statements: dictionary,
  default-edge-statements: dictionary,
  default-node-data: any,
  default-edge-data: any,
  default-source-data: any,
  default-sink-data: any
) -> dictionary
```

12.2.1.1.1 ..items `array`

Node and edge items returned by `node` and `edge`. The best way to use these is to use a code scope, so that the edges and nodes append each other:

```
#let g = build({
  node(<a>, label: [a])
  node(<b>, label: [b])
  node(<c>, label: [c])
  edge(source(<a>), sink(<b>))
  edge(source(<b>), sink(<c>))
  edge(source(<a>), sink(<c>))
})
```

12.2.1.1.2 name `none` or `string`

Graph name.

```
#let g = build(name: "My Graph", {
})
#info(g).name
```

Default: `none`

12.2.1.1.3 data `any`

Native Typst graph data.

```
#let g = build(data: (a:(b:(1, ))), {
})
#info(g).data
```

Default: `none`

12.2.1.1.4 statements `dictionary`

Flat graph statements, that get turned into a string to string dictionary in rust. Used by DOT parsing; values cannot nest.

```
#let g = build(statements: (a:1), {
})
#info(g).global-statements
```

Default: `(:)`

12.2.1.1.5 default-node-statements `dictionary`

Flat default node statements. Used by DOT; values cannot nest.

Default: `(:)`

12.2.1.1.6 default-edge-statements `dictionary`

Flat default edge statements. Used by DOT parsing; values cannot nest. These are applied/delegated to all edges.

```
#let g = build(default-edge-statements: (a:1), {
   node(<a>)
   edge(source(<a>))
   edge(sink(<a>))
})
#edges(g).map(e=>e.statements)
```

Default: `(:)`

12.2.1.1.7 default-node-data `any`

Default data merged into every node data. Captured node data fields override it.

```
#let g = build(default-node-data: (a:1), {
   node(<a>)
   node(<b>,a:2)
   node(<c>,b:(a:1))
})
#nodes(g).map(n=>n.data)
```

Default: `none`

12.2.1.1.8 default-edge-data `any`

Default data merged into every edge data. Captured edge data fields override it.

```
#let g = build(default-edge-data: (a:1,b:[$m_mu$]), {
   node(<a>,id:0)
   edge(source(<a>),id:1)
   edge(sink(<a>),<e>,id:0,b:[#set text(font:"Reforma")
   This is a test: ])
})
#edges(g).map(e=>e.data.b).join()
```

Default: `none`

12.2.1.1.9 default-source-data `any`

Default data merged into every source half-edge data. Captured source data fields override it.

```
#let g = build(default-source-data: (a:1), {
   node(<a>)
   edge(source(<a>))
   edge(sink(<a>))
})
#edges(g).map(e=>if e.source != none {e.source.data} else {none})
```

Default: `none`

12.2.1.1.10 default-sink-data `any`

Default data merged into every sink half-edge data. Captured sink data fields override it.

```
#let g = build(default-sink-data: (a:1), {
   node(<a>)
   edge(source(<a>))
   edge(sink(<a>))
})
#edges(g).map(e=>if e.sink != none {e.sink.data} else {none})
```

Default: `none`

12.2.2 parse

Parse one or more DOT graphs into graph objects.

Default data are applied before `eval-*` fields are evaluated, so default data strings can refer to parsed record fields such as `#str(name)`. Parsed fields take precedence over default data fields, so DOT `label="..."` overrides `default-node-data: (label: ...)`.

Linnest uses `dot-parser` for DOT syntax and then gives special meaning to a small set of attributes. Every other attribute is preserved as a flat string statement. Preserved statements are visible through `info`, `nodes`, `edges`, `map`, and `eval-fields`, but they do not become native Typst `data` unless a matching `eval-*` argument is passed.

````
#let src = ```dot
digraph { a [label="A"]; }
```
#let n = nodes(parse(src.text).first()).first()
#n.statements.label
#n.data
````

Graph-level handling:

- The DOT graph name becomes `graph.info(g).name`. A graph-level `name` attribute can name an anonymous graph; a named `graph`/`digraph` header takes precedence.

````
#let src = ```dot
digraph header_wins { graph [name="ignored"]; a }
```
#info(parse(src.text).first()).name
````

- Top-level `key=value` statements and `graph [key=value]` attributes become `graph.info(g).global-statements`, except for the consumed `name`. They do not configure layout, even when a key looks like a layout option; pass layout options directly to `layout`.

````
#let src = ```dot
digraph { steps=1; graph [subtitle="nested"]; a }
```
#let statements = info(parse(src.text).first()).global-statements
#statements.steps
#statements.subtitle
````

- `node [key=value]` and `edge [key=value]` become `default-node-statements` and `default-edge-statements`. They are merged into each parsed node or edge before local attributes, so local attributes override defaults.

````
#let src = ```dot
digraph { node [color=gray]; edge [particle=g]; a [color=red]; a -> b }
```
#let g = parse(src.text).first()
#nodes(g).first().statements.color
#edges(g).first().statements.particle
````

Node handling:

- The DOT node id becomes the node `name` returned by `nodes`. If the DOT node id is numeric, it is consumed as the node index instead and the public `name` is `none`.

````
#let src = ```dot
digraph { alpha; 1; }
```
#nodes(parse(src.text).first()).map(n => n.name)
````

- `id=<n>` is also consumed as an explicit node index. Explicit node indexes must be unique and in bounds after parsing.
````

```
#let src = ```dot
digraph { a [id=1]; b [id=0]; }
```
#nodes(parse(src.text).first()).map(n => n.name)
```

- `style=invis` marks the DOT node as a dangling external endpoint. It is not returned by `nodes`. Its remaining attributes are copied onto the external edge that touches it, except `shape` and `label`; `style` and `id` are consumed.

```
#let src = ```dot
digraph { ext [style=invis, column=left, label=skip]; ext -> a [id=0]; }
```
#let g = parse(src.text).first()
#nodes(g).map(n => n.name)
#edges(g).first().statements.column
#edges(g).first().statements.at("label", default: none)
```

- Node `style` is consumed for every node; only `style=invis` has special behavior. Use another attribute name for drawing style metadata that must survive parsing.

```
#let src = ```dot
digraph { a [style=filled, "draw-style"=filled]; }
```
#nodes(parse(src.text).first()).first().statements
```

- `pos` is parsed as node placement. Simple numeric values are exposed as `node.pos`; extended placement values are used by `layout`. `pin` is an explicit layout constraint and is not exposed as a public statement.

```
#let src = ```dot
digraph { a [id=0, pos="0,0!"]; b [id=1, pos="ref(node:0)+2,0!"]; }
```
#let parsed-nodes = nodes(parse(src.text).first())
#parsed-nodes.map(n => n.pos)
#parsed-nodes.map(n => n.statements.at("pin", default: none))
```

- `shift` is parsed as a drawing shift and also remains available as a statement. `eval` is retained for legacy round-tripping. Other node attributes are preserved as node statements.

```
#let src = ```dot
digraph { a [shift="0.1,0", eval=legacy, label=A]; }
```
#let n = nodes(parse(src.text).first()).first()
#n.shift
#n.statements
```

Edge handling:

- `id=&lt;n&gt;` is consumed as the edge index. It is not preserved as a statement; drawing callbacks expose the resulting stable edge index as `eid`. Explicit edge indexes must be unique and in bounds. Without explicit ids, edge order is parser-internal, not DOT input order.

```
#let src = ```dot
digraph { a -> b [id=1, label=later]; b -> c [id=0, label=first]; }
```
#let parsed-edges = edges(parse(src.text).first())
```

```
#parsed-edges.map(e => e.edge)
#parsed-edges.map(e => e.statements.label)
#parsed-edges.map(e => e.statements.at("id", default: none))
```

- `dir=forward`, `dir=back`, and `dir=none` become edge orientations `"default"`, `"reversed"`, and `"undirected"`. If omitted, directed DOT edges are `"default"` and undirected DOT edges are `"undirected"`.

````
#let src = ```dot
digraph { a -> b [id=0]; b -> c [id=1, dir=back]; c -> d [id=2, dir=none]; }
```
#edges(parse(src.text).first()).map(e => e.orientation)
````

- `source="..."` and `sink="..."` are consumed as endpoint `statement` values and removed from edge statements. On an external edge, use the attribute matching the real endpoint side in the DOT edge: `ext -> a` uses `sink=...`, while `a -> ext` uses `source=...`.

````
#let src = ```dot
digraph { ext [style=invis]; ext -> a [id=0, sink=in]; a -> ext [id=1, source=out]; }
```
#let endpoint-statement(endpoint) = if endpoint == none { none } else
{ endpoint.at("statement", default: none) }
#let parsed-edges = edges(parse(src.text).first())
#parsed-edges.map(e => endpoint-statement(e.source))
#parsed-edges.map(e => endpoint-statement(e.sink))
#parsed-edges.map(e => e.statements.at("source", default: none))
````

- `pos` is parsed as the edge control-point placement. `pin` is an explicit edge layout constraint. `shift`, `label-pos`, `label-angle`, and `bend` are parsed into edge geometry fields; except for `pin`, they also remain available as statements. Other edge attributes are preserved as edge statements.

````
#let src = ```dot
digraph { a -> b [id=0, pos="0,1!", pin="x:@edge", shift="0.1,0", "label-
pos"="0,1.2", "label-angle"="0.3rad", bend="0.4rad", particle=g]; }
```
#let e = edges(parse(src.text).first()).first()
#e.pos
#e.shift
#e.label-pos
#e.label-angle
#e.bend
#e.statements.particle
#e.statements.at("pin", default: none)
````

- Attributes copied from an invisible endpoint node are merged into the external edge statements unless their keys are `shape` or `label`.

````
#let src = ```dot
digraph { ext [style=invis, column=left, shape=none, label=skip]; ext -> a [id=0]; }
```
#edges(parse(src.text).first()).first().statements
````

Port and half-edge handling:

- DOT ports of the form `node:port` and `node:port:compass` are preserved on source/sink endpoint records. Numeric ports also assign explicit half-edge ids. Non-numeric ports are exposed only as `port-label`.

```
#let src = ```dot
digraph { a:left:e -> b:0:w [id=0]; }
```
#let e = edges(parse(src.text).first()).first()
#e.source.port-label
#e.sink.hedge
```

- Compass values are exposed as `compass`; valid DOT compass names include `n`, `ne`, `e`, `se`, `s`, `sw`, `w`, `nw`, `c`, and `_`.

```
#let src = ```dot
digraph { a:n -> b:sw [id=0]; }
```
#let e = edges(parse(src.text).first()).first()
#e.source.compass
#e.sink.compass
```

- Explicit half-edge ids from numeric ports must be unique and in bounds. They are retained for half-edge ordering and for `join` with `key: "id"`; query and eval records expose the resulting half-edge index as `hedge`.

```
#let src = ```dot
digraph { a:1 -> b:0 [id=0]; }
```
#let e = edges(parse(src.text).first()).first()
#e.source.hedge
#e.sink.hedge
```

Placement fields:

- `pos="x,y"` gives a starting coordinate; `pos="x,y!"` pins both axes.

```
#let src = ```dot
digraph { a [id=0, pos="0,0"]; b [id=1, pos="2,0!"]; }
```
#let parsed-nodes = nodes(parse(src.text).first())
#parsed-nodes.map(n => n.pos)
#parsed-nodes.map(n => n.statements.at("pos-mode", default: none))
```

- `pos="ref(node:<id>)+dx,dy!"` and `pos="ref(edge:<id>)+dx,dy!"` reference explicit DOT node or edge ids, not names and not implicit parse order.

```
#let src = ```dot
digraph { a [id=0, pos="1,1!"]; b [id=1, pos="ref(node:0)+2,0!"]; a -> b [id=0,
pos="ref(node:1)+0,1!"]; }
```
#let g = parse(src.text).first()
#nodes(g).at(1).pos
#edges(g).first().pos
```

- Axis form accepts `x:&lt;coord&gt;` and `y:&lt;coord&gt;` entries. `!` applies to the individual axis, for example `pos="x:2!,y:0"`.

```
#let src = ```dot
digraph { a [id=0, pos="x:2!,y:0"]; }
```
```

```
#nodes(parse(src.text).first()).first().pos
```

- Group coordinates use `@name`, `@+name`, or `@-name` in axis form and must be pinned with `!`, for example `pos="x:@-left!,y:@row!"`.

```
#let src = ```dot
digraph { ext [style=invis]; ext -> a [id=0, pos="x:@-left!,y:@row!"]; }
```
#edges(layout(parse(src.text).first(), layout-algo: "tree")).first().pos
```

- `pin` accepts numeric point constraints, `x:<coord>`, `y:<coord>`, and grouped constraints such as `x:@left`, `x:@+right`, or `@row`.

```
#let src = ```dot
digraph { a -> b [id=0, pin="x:@+right"]; }
```
#edges(layout(parse(src.text).first(), layout-algo:
"tree")).first().statements.at("pin", default: none)
```

12.2.2.1 Parameters

```
parse(
  input: string bytes,
  default-node-data: any,
  eval-node-fields: string array,
  default-edge-data: any,
  eval-edge-fields: string array,
  default-source-data: any,
  eval-source-fields: string array,
  default-sink-data: any,
  eval-sink-fields: string array,
  eval-graph-fields: string array,
  eval-mode: string,
  scope: dictionary
) -> array
```

12.2.2.1.1 input `string` or `bytes`

DOT source text containing one or more `graph` or `digraph` definitions.

12.2.2.1.2 default-node-data `any`

Default data merged into every node data. Captured node data fields override it, but bare dot statements don't set data fields. To turn statements into data fields, use `eval-node-fields`.

````
#let a = ```dot
digraph {
a -> b -> c -> d -> a
a -> a
b -> d
c [particle="q"]
}
```
#let g = parse(a.text,default-node-data:(particle:"g")).at(0)
#nodes(g).map(n=>n.data.particle)
````

Default: `none`

12.2.2.1.3 eval-node-fields `string` or `array`

Node fields to evaluate into node data. The eval scope includes node statements plus `node`, `name`, and `pos`.

````
#let src = ```dot
digraph { a [label="#str(name)"]; }
```
#nodes(parse(src.text, eval-node-fields: "label").first()).first().data.label
````

Default: `()`

12.2.2.1.4 default-edge-data `any`

Default data merged into every edge data. Captured edge data fields override it, but bare dot statements don't set data fields. To turn statements into data fields, use `eval-edge-fields`.

Default: `none`

12.2.2.1.5 eval-edge-fields `string` or `array`

Edge statement fields to evaluate into `graph.edges(g).at(i).data`. The eval scope includes edge statements plus `edge`, `orientation`, endpoint records, placement fields, and existing edge data. Drawing callbacks later expose the edge index as `eid`.

````
#let src = ```dot
digraph { a -> b [id=0, label="#orientation"]; }
```
#edges(parse(src.text, eval-edge-fields: "label").first()).first().data.label
````

Default: `()`
````

12.2.2.1.6 default-source-data `any`

Default data merged into every source half-edge data. Captured source data fields override it.

Default: `none`

12.2.2.1.7 eval-source-fields `string` or `array`

Source half-edge fields to evaluate into `edge.source.data`. The eval scope includes source endpoint fields `statement`, `port-label`, `compass`, `node`, and `hedge`, plus the surrounding edge fields.

```
#let src = ```dot
digraph { a:left:e -> b [id=0, source="#port-label"]; }
```
#edges(parse(src.text, eval-source-fields:
"statement").first()).first().source.data.statement
```

Default: `()`

#### 12.2.2.1.8 default-sink-data `any`

Default data merged into every sink half-edge data. Captured sink data fields override it.

Default: `none`

#### 12.2.2.1.9 eval-sink-fields `string` or `array`

Sink half-edge fields to evaluate into `edge.sink.data`. The eval scope includes sink endpoint fields `statement`, `port-label`, `compass`, `node`, and `hedge`, plus the surrounding edge fields.

```
#let src = ```dot
digraph { a -> b:0:w [id=0, sink="#str(hedge)"]; }
```
#edges(parse(src.text, eval-sink-fields:
"statement").first()).first().sink.data.statement
```

Default: `()`

12.2.2.1.10 eval-graph-fields `string` or `array`

Graph statement fields to evaluate into `graph.info(g).data`. The eval scope includes graph statements and direct graph record fields.

```
#let src = ```dot
digraph {
  title = "Strong graph";
}
```
#info(parse(src.text, eval-graph-fields: "title").first()).data.title
```

Default: `()`

#### 12.2.2.1.11 eval-mode `string`

Typst `eval` mode used for string field values.

Default: `"markup"`

#### 12.2.2.1.12 scope `dictionary`

Additional Typst names available while evaluating field values.

Default: `(:)`

### 12.2.3 node

Create a graph node item for `build`.

A Typst label is the node name used by `source`, `sink`, and `pos`. The optional numeric `id` fixes the resulting graph node index. Extra named arguments are captured as node data fields, so `node(<a>, label: [A], color: red)` stores `(label: [A], color: red)`. The default draw style uses `data.label` as the visible node label when present.

#### 12.2.3.1 Parameters

```
node(
  ..args: label,
  name: none label,
  id: none int,
  pos: none dictionary,
  shift: none string array dictionary,
  statements: dictionary
) -> array
```

12.2.3.1.1 ..args `label`

Optional positional node name. Must be a Typst label when provided; extra named arguments become data fields.
```

12.2.3.1.2 name `none` or `label`

Typst node name for references.

Default: `none`

12.2.3.1.3 id `none` or `int`

Numeric graph node index. Must be unique and in bounds when provided.

Default: `none`

12.2.3.1.4 pos `none` or `dictionary`

Node placement.

Default: `none`

12.2.3.1.5 shift `none` or `string` or `array` or `dictionary`

Drawing shift stored as a statement.

Default: `none`

12.2.3.1.6 statements `dictionary`

Additional flat node statements. Used by DOT; values cannot nest.

Default: `(:)`

12.2.4 source

Create a source half-edge endpoint.

`node` may be a node name like `<a>` or a numeric node index. `name` gives the half-edge a Typst name; `id` is a numeric half-edge order/index override. Extra named arguments are captured as source data fields.

12.2.4.1 Parameters

```
source(
  node: label int,
  ..args: any,
  name: none label,
  id: none int,
  statement: none string,
  compass: none string
) -> dictionary
```

12.2.4.1.1 node `label` or `int`

Referenced node, either by Typst label name or numeric node index.

12.2.4.1.2 ..args `any`

Extra named arguments become source data fields.

12.2.4.1.3 name `none` or `label`

Optional half-edge name used for later references.

Default: `none`

12.2.4.1.4 id `none` or `int`

Optional numeric half-edge index/order override.

Default: `none`

12.2.4.1.5 statement `none` or `string`

DOT-ish flat statement used for matching/joining dangling half edges.

Default: `none`

12.2.4.1.6 compass `none` or `string`

DOT compass point such as `"n"`, `"s"`, `"e"`, or `"w"`.

Default: `none`

12.2.5 sink

Create a sink half-edge endpoint.

`node` may be a node name like `<a>` or a numeric node index. `name` gives the half-edge a Typst name; `id` is a numeric half-edge order/index override. Extra named arguments are captured as sink data fields.

12.2.5.1 Parameters

```
sink(
  node: label int,
  ..args: any,
  name: none label,
  id: none int,
  statement: none string,
  compass: none string
) -> dictionary
```

12.2.5.1.1 node `label` or `int`

Referenced node, either by Typst label name or numeric node index.

12.2.5.1.2 ..args `any`

Extra named arguments become sink data fields.

12.2.5.1.3 name `none` or `label`

Optional half-edge name used for later references.

Default: `none`

12.2.5.1.4 id `none` or `int`

Optional numeric half-edge index/order override.

Default: `none`

12.2.5.1.5 statement `none` or `string`

DOT-ish flat statement used for matching/joining dangling half edges.

Default: `none`

12.2.5.1.6 compass `none` or `string`

DOT compass point such as `"n"`, `"s"`, `"e"`, or `"w"`.

Default: `none`

12.2.6 edge

Create a graph edge item for `build`.

Positional arguments may contain one `source`, one `sink`, and optionally one Typst label used as the edge name. The numeric `id` chooses the edge order. Extra named arguments are captured as edge data fields, so `edge(source(<a>), sink(<b>), particle: "g")` stores `(particle: "g")`. The default draw style uses `data.label` as the visible edge label when present.

12.2.6.1 Parameters

```
edge(
  ..args: any,
  name: none label,
  id: none int,
  orientation: string,
  pos: none dictionary,
  shift: none string array dictionary,
  label-pos: none string array dictionary,
  label-angle: none int float string,
  bend: none int float string,
  statements: dictionary
) -> array
```

12.2.6.1.1 ..args `any`

Source/sink half-edges and optional edge name; extra named arguments become data fields.

12.2.6.1.2 name `none` or `label`

Typst edge name.

Default: `none`

12.2.6.1.3 id `none` or `int`

Numeric edge order/index override.

Default: `none`

12.2.6.1.4 orientation `string`

Edge orientation: `"default"`, `"reversed"`, or `"undirected"`.

Default: `"default"`

12.2.6.1.5 pos `none` or `dictionary`

Edge placement.

Default: `none`

12.2.6.1.6 shift `none` or `string` or `array` or `dictionary`

Drawing shift stored as a statement.

Default: `none`

12.2.6.1.7 label-pos `none` or `string` or `array` or `dictionary`

Edge label position stored as a statement.

Default: `none`

12.2.6.1.8 label-angle `none` or `int` or `float` or `string`

Edge label angle stored as a statement.

Default: `none`

12.2.6.1.9 bend `none` or `int` or `float` or `string`

Edge bend stored as a statement.

Default: `none`

12.2.6.1.10 statements `dictionary`

Additional flat edge statements. Used by DOT; values cannot nest.

Default: `(:)`

12.2.7 group

Create a grouped placement coordinate.

`side: "+"` keeps the solved coordinate non-negative and `side: "-"` keeps it non-positive. Groups are layout constraints and therefore require pin placement, which is the `pos` default.

```
#group("right", side: "+")
```

12.2.7.1 Parameters

```
group(
  name: string int bool,
  side: none string
) -> dictionary
```

12.2.7.1.1 name `string` or `int` or `bool`

Group identifier shared by positions constrained to the same coordinate.

12.2.7.1.2 side `none` or `string`

Optional sign constraint: `"+"`, `"-"`, `"positive"`, or `"negative"`.

Default: `none`

12.2.8 pin

Mark one coordinate as a layout constraint.

12.2.8.1 Parameters

`pin(value: int float dictionary) -> dictionary`

12.2.8.1.1 value `int` or `float` or `dictionary`

Coordinate value to constrain.

12.2.9 start

Mark one coordinate as an initial layout value only.

12.2.9.1 Parameters

`start(value: int float) -> dictionary`

12.2.9.1.1 value `int` or `float`

Coordinate value to use as the layout seed.

12.2.10 pos

Create a first-class graph placement.

The default `mode: "pin"` turns numeric coordinates into layout constraints and also makes the coordinates immediately drawable without a layout pass. Use `start(value)` for an individual coordinate that should only seed the layout, or `pin(value)` to pin an individual numeric coordinate when `mode: "start"` is used. Grouped coordinates are always layout constraints for their axis.

```
#pos(x: group("right", side: "+"), y: start(10))
```

12.2.10.1 Parameters

```
pos(
  x: none int float dictionary,
  y: none int float dictionary,
  ref: none label int,
  dx: none int float,
  dy: none int float,
  mode: string
) -> dictionary
```

12.2.10.1.1 x `none` or `int` or `float` or `dictionary`

Absolute or grouped x coordinate.

Default: `none`

12.2.10.1.2 y `none` or `int` or `float` or `dictionary`

Absolute or grouped y coordinate.

Default: `none`

12.2.10.1.3 ref `none` or `label` or `int`

Node reference for relative placement, by name or numeric index.

Default: `none`

12.2.10.1.4 dx `none` or `int` or `float`

Relative x offset from `ref`.

Default: `none`

12.2.10.1.5 dy `none` or `int` or `float`

Relative y offset from `ref`.

Default: `none`

12.2.10.1.6 mode `string`

Placement mode: `"pin"` constrains layout, `"start"` only seeds it.

Default: `"pin"`

12.2.11 map

Map graph metadata to new native data.

The callbacks receive decoded records plus a `fields` dictionary containing merged statements and direct record fields. A callback returns `none` to leave the record unchanged, `(data: value)` to set new native data, or structural fields such as `pos`, `shift`, and `statements` to patch data seen by later layout calls.

```
#let g = build({ node(<a>) })
#let g = map(g, node: node => (data: (label: [A])))
#nodes(g).first().data.label
```

12.2.11.1 Parameters

```
map(
  graph_: dictionary,
  graph: none function,
  node: none function,
  edge: none function,
  source: none function,
  sink: none function
) -> dictionary
```

12.2.11.1.1 graph_ `dictionary`

Graph object to transform.

12.2.11.1.2 graph `none` or `function`

Callback for graph metadata records.

Default: `none`

12.2.11.1.3 node `none` or `function`

Callback for node records.

Default: `none`

12.2.11.1.4 edge `none` or `function`

Callback for edge records.

Default: `none`

12.2.11.1.5 source `none` or `function`

Callback for source half-edge records.

Default: `none`

12.2.11.1.6 sink `none` or `function`

Callback for sink half-edge records.

Default: `none`

12.2.12 style

Attach layout-relevant drawing style to a graph.

Node style is measured immediately and stored as `layout-width` / `layout-height` statements for later layout calls. Edge labels are measured as `label-width` / `label-height` statements for label placement. `draw` uses the stored node and edge-label style by default.

12.2.12.1 Parameters

```
style(
  graph_: dictionary,
  scope: dictionary,
  unit: int float length ratio,
  node-label: auto content string function none,
  node-label-style: dictionary function,
  node-style: dictionary function none,
  edge-label: content string function none,
  edge-label-style: dictionary function
) -> dictionary
```

12.2.12.1.1 graph_ `dictionary`

Graph object to style.

12.2.12.1.2 scope `dictionary`

Extra scope visible to label/style callbacks.

Default: `(:)`

12.2.12.1.3 unit `int` or `float` or `length` or `ratio`

Coordinate length for one graph-layout unit. Numbers are interpreted as em.

Default: `1`

12.2.12.1.4 node-label `auto` or `content` or `string` or `function` or `none`

Node label content or callback. `auto` uses node data/statement label or node name.

Default: `auto`

12.2.12.1.5 node-label-style `dictionary` or `function`

CeTZ content style for node labels. Its padding contributes to measured node size.

Default: `(:)`

12.2.12.1.6 node-style `dictionary` or `function` or `none`

CeTZ node shape style or callback. Explicit numeric radii contribute to measured node size.

Default: `(:)`

12.2.12.1.7 edge-label `content` or `string` or `function` or `none`

Edge label content or callback. `none` leaves edge labels unstyled and unmeasured.

Default: `none`

12.2.12.1.8 edge-label-style `dictionary` or `function`

CeTZ content style for edge labels. Its padding contributes to measured edge-label size.

Default: `(:)`

12.2.13 eval-fields

Evaluate selected fields into native data entries.

Each selected field is read from the record's merged `fields` dictionary, evaluated in a scope containing those fields, and written to `data.<field>`.

```
#let g = build({
  node(<a>)
  node(<b>)
  edge(source(<a>), sink(<b>), statements: (mom: "p"))
}, default-edge-statements: (display-label: "$#mom$"))
#let g = eval-fields(g, eval-edge-fields: ("display-label",))
#edges(g).first().data.at("display-label")
```

12.2.13.1 Parameters

```
eval-fields(
  graph_: dictionary,
  eval-graph-fields: string array,
  eval-node-fields: string array,
  eval-edge-fields: string array,
  eval-source-fields: string array,
  eval-sink-fields: string array,
  eval-mode: string,
  scope: dictionary
) -> dictionary
```

12.2.13.1.1 graph_ `dictionary`

Graph object whose selected statement fields should be evaluated.

12.2.13.1.2 eval-graph-fields `string` or `array`

Graph statement fields to evaluate into `graph.info(g).data`.

Default: `()`

12.2.13.1.3 eval-node-fields `string` or `array`

Node statement fields to evaluate into node data.

Default: `()`

12.2.13.1.4 eval-edge-fields `string` or `array`

Edge statement fields to evaluate into edge data.

Default: `()`

12.2.13.1.5 eval-source-fields `string` or `array`

Source half-edge fields to evaluate into source data.

Default: `()`

12.2.13.1.6 eval-sink-fields `string` or `array`

Sink half-edge fields to evaluate into sink data.

Default: `()`

12.2.13.1.7 eval-mode `string`

Typst `eval` mode used for string field values.

Default: `"markup"`

12.2.13.1.8 scope `dictionary`

Additional Typst names available while evaluating field values.

Default: `(:)`

12.2.14 info

Return graph metadata.

The result has `name`, `global-statements`, `default-edge-statements`, and `default-node-statements`.

```
#let g = build({ node(<a>) }, name: "demo")
#info(g).name
```

12.2.14.1 Parameters

`info(graph: dictionary) -> dictionary`

12.2.14.1.1 graph `dictionary`

Graph object returned by `build`, `parse`, `layout`, or another graph API.

12.2.15 dot

Serialize a graph object to DOT.

```
#let g = build({
  node(<a>)
  node(<b>)
  edge(source(<a>), sink(<b>))
}, name: "demo")
#dot(g).contains("digraph demo")
```

12.2.15.1 Parameters

`dot(graph: dictionary) -> string`

12.2.15.1.1 graph `dictionary`

Graph object to serialize.

12.2.16 nodes

Return node records, optionally filtered by a subgraph object.

Node `name` values are Typst labels when present.

```
#let g = build({
  node(<a>)
  node(<b>)
})
#nodes(g).map(node => str(node.name)).join(", ")
```

12.2.16.1 Parameters

```
nodes(
  graph: dictionary,
  subgraph: none bytes
) -> array
```

12.2.16.1.1 graph `dictionary`

Graph object to inspect.

12.2.16.1.2 subgraph `none` or `bytes`

Optional subgraph filter; only nodes incident to selected half edges are returned.

Default: `none`

12.2.17 edges

Return edge records, optionally filtered by a subgraph object.

Edge `name` values are Typst labels when present.

```
#let g = build({
  node(<a>)
  node(<b>)
  edge(source(<a>, compass: "e"), sink(<b>))
})
#edges(g).len()
```

12.2.17.1 Parameters

```
edges(
  graph: dictionary,
  subgraph: none bytes
) -> array
```

12.2.17.1.1 graph `dictionary`

Graph object to inspect.

12.2.17.1.2 subgraph `none` or `bytes`

Optional subgraph filter; only selected edges/half-edges are returned.

Default: `none`

12.2.18 node-data

Return one named node's native data.

`name` is a Typst label such as `<a>` or the corresponding string name.

```
#let g = build({ node(<a>, label: [A]) })
#node-data(g, <a>).label
```

12.2.18.1 Parameters

```
node-data(
  graph_: dictionary,
  name: label string
) -> any
```

12.2.18.1.1 graph_ `dictionary`

Graph object to inspect.

12.2.18.1.2 name `label` or `string`

Node name as a Typst label or its string form.

12.2.19 edge-data

Return one named edge's native data.

`name` is a Typst label such as `<e>` or the corresponding string name.

```
#let g = build({
  node(<a>)
  node(<b>)
  edge(source(<a>), <e>, sink(<b>), label: [$p$])
})
#edge-data(g, <e>).label
```

12.2.19.1 Parameters

```
edge-data(
  graph_: dictionary,
  name: label string
) -> any
```

12.2.19.1.1 graph_ `dictionary`

Graph object to inspect.

12.2.19.1.2 name `label` or `string`

Edge name as a Typst label or its string form.

12.2.20 update-node-data

Update one named node's native data.

`update` may be a replacement data value or a function `(data, node) => new-data`.

```
#let g = build({ node(<a>) })
#let g = update-node-data(g, <a>, (label: [A]))
#nodes(g).first().data.label
```

12.2.20.1 Parameters

```
update-node-data(
  graph_: dictionary,
  name: label string,
  update: any function
) -> dictionary
```

12.2.20.1.1 graph_ `dictionary`

Graph object to update.

12.2.20.1.2 name `label` or `string`

Node name as a Typst label or its string form.

12.2.20.1.3 update `any` or `function`

Replacement data or `(data, node) => new-data` callback.

12.2.21 update-edge-data

Update one named edge's native data.

`update` may be a replacement data value or a function `(data, edge) => new-data`.

```
#let g = build({
  node(<a>)
  node(<b>)
  edge(source(<a>), <e>, sink(<b>))
})
#let g = update-edge-data(g, <e>, (label: [$p$]))
#edges(g).first().data.label
```

12.2.21.1 Parameters

```
update-edge-data(
  graph_: dictionary,
  name: label string,
  update: any function
) -> dictionary
```

12.2.21.1.1 graph_ `dictionary`

Graph object to update.

12.2.21.1.2 name `label` or `string`

Edge name as a Typst label or its string form.

12.2.21.1.3 update `any` or `function`

Replacement data or `(data, edge) => new-data` callback.

12.2.22 join

Join two graphs by matching dangling half-edge statements or ids on `key`.

Supported key values are `"statement"`, `"compass"`, and `"id"`.

```
#let left = build({
  node(<a>)
  edge(sink(<a>, statement: "j"))
})
#let right = build({
  node(<b>)
  edge(source(<b>, statement: "j"))
})
#edges(join(left, right, key: "statement")).len()
```

12.2.22.1 Parameters

```
join(
  left: dictionary,
  right: dictionary,
  key: string
) -> dictionary
```

12.2.22.1.1 left `dictionary`

Left graph object.

12.2.22.1.2 right `dictionary`

Right graph object.

12.2.22.1.3 key `string`

Dangling half-edge match key: `"statement"`, `"compass"`, or `"id"`.

Default: `"statement"`

12.2.23 cycles

Return subgraph objects for the graph's cycle basis.

```
#let g = build({
  node(<a>)
  node(<b>)
  edge(source(<a>), sink(<b>))
})
#cycles(g).len()
```

12.2.23.1 Parameters

```
cycles(graph: dictionary) -> array
```

12.2.23.1.1 graph `dictionary`

Graph object to analyze.

12.2.24 forests

Return subgraph objects for the graph's spanning forests.

```
#let g = build({
  node(<a>)
  node(<b>)
  edge(source(<a>), sink(<b>))
})
#forests(g).len()
```

12.2.24.1 Parameters

`forests(graph: dictionary) -> array`

12.2.24.1.1 graph `dictionary`

Graph object to analyze.

13 Layout functions

`layout` and `layout-sequence` are top-level functions exported by `lib.typ`, not members of a public `layout` module. Their generated entries come from the shared `layout.typ` implementation and use the public export names.

13.1 Concepts

13.1.1 Placements

`graph.pos` creates a first-class placement. The default `mode: "pin"` turns a coordinate into a fixed layout constraint and a drawable position. Use `mode: "start"` when the coordinate should only seed the layout:

```
#let g = build({
  node(<a>, pos: graph.pos(x: -2, y: 0))
  node(<c>, pos: graph.pos(ref: <a>, dx: 4, dy: 0))
  edge(source(<a>), <a-c>, sink(<c>), pos: graph.pos(x: 0, y: 1.2))
})
```

`graph.group` links one coordinate across several nodes or edge control points. A `side` of `"+"` keeps the coordinate positive, and `"-"` keeps it negative. GammaLoop external-edge columns use this to keep incoming and outgoing external legs on opposite sides while pairing rows by a shared `y` group:

```
#let edge-items = (
  edge(
    source(<right-ext>),
```

```
    sink(<center>),
    pos: graph.pos(x: graph.group("right", side: "+"), y: graph.group("edgee0")),
  ),
  edge(
    source(<center>),
    sink(<left-ext>),
    pos: graph.pos(x: graph.group("left", side: "-"), y: graph.group("edgee0")),
  ),
)
```

Raw DOT input uses the same placement model through the `pos` attribute. The standard Graphviz subset is preserved: `pos="x,y"` is a starting coordinate and `pos="x,y!"` is pinned. Linnest extends the value with explicit id references and axis entries. In axis entries, `!` belongs to that axis: numeric entries without `!` are starting coordinates, numeric entries with `!` are fixed constraints, and grouped entries must use `!` because groups are constraints.

```
digraph {
  a [id=0 pos="0,0!"]
  b [id=1 pos="ref(node:0)+4,0!"]
  a -> b [id=0 pos="ref(node:1)+0,1!"]
  b -> c [id=1 pos="ref(edge:0)+1,0!"]
  c [id=2 pos="x:2!"]
  d [id=3 pos="x:2!,y:1"]
  ext -> a [id=2 pos="x:@-left!,y:@edge0!"]
}
```

The DOT grammar is intentionally id-based: `ref(node:0)` uses a node `id` and `ref(edge:0)` uses an edge `id`. Bare names and implicit edge order are not placement references. The `@` syntax denotes grouped coordinate constraints; `@+name` and `@-name` keep the grouped coordinate on the positive or negative side respectively.

```
pos="x,y"                   // start x and y
pos="x,y!"                  // pin x and y
pos="x:<coord>"             // start numeric x
pos="x:<coord>!"            // pin x
pos="y:<coord>!"            // pin y
pos="x:<coord>!,y:<coord>"  // pin x, start numeric y
pos="x:<coord>,y:<coord>!"  // start numeric x, pin y
```

13.1.2 Layout Model

`layout` starts from a traversal-tree placement, then optimizes the positions of graph nodes and edge control points. The initial tree spacing is

$$L = \lambda \sqrt{\frac{WH}{\max(n, 1)}}$$

,

with horizontal spacing $\tau_x L$ and vertical spacing $\tau_y L$. Here λ is `length-scale`, W is `viewport-w`, H is `viewport-h`, τ_x is `tree-dx`, and τ_y is `tree-dy`. These fields set the geometry scale for both layout modes.

For deterministic, non-iterative placement, use `layout-algo: "tree"` or `layout-algo: "dot"`. `"tree"` places a traversal forest by levels. `"dot"` uses a directed layered placement for acyclic inputs, falling back to tree placement when the selected edges contain a cycle. Both modes also work on a subgraph:

```
#let g = parse("digraph partial { a -> b; b -> c; c -> d; d -> a }").at(0)
#let tree = graph.forests(g).at(0)
```

```
#let g = layout(g, layout-algo: "tree", subgraph: tree)
#draw(g)
```

Only nodes touched by the selected subgraph are placed. Edge control points are then resolved from the current node positions, so edges outside the selected subgraph are drawn as straight lines unless their own position constraints say otherwise.

`layout-algo: "force"` and `layout-algo: "anneal"` also accept `subgraph`. For these iterative modes, nodes and edge control points outside the selected subgraph stay fixed and act as boundary points while the selected subgraph is optimized.

Set `layout-nodes: "fixed"` to keep every node at its current `pos` for this layout pass and move only edge control points. With `subgraph`, only edges in the selected subgraph are moved; all other edge control points keep their current positions. The fixed-node policy is temporary; the returned graph stores the resulting coordinates as `pos`, but it does not turn them into persistent `pin` constraints. This is useful after one node-placement pass when a later pass should route or relax selected edges without disturbing the node layout:

```
#let g = parse("digraph partial { a [pos=\"0,0\"]; b [pos=\"4,0\"]; c [pos=\"8,0\"]; a ->
b; b -> c }").at(0)
#let first = subgraph.bits(g, (true, true, false, false))
#let g = layout(g, layout-algo: "tree", layout-nodes: "fixed", subgraph: first)
#draw(g)
```

The shared spring/charge model uses the following coefficients:

$$c_{\mathrm{vv}} = \beta L^2$$

$$c_{\mathrm{ev}} = \beta \gamma_{\mathrm{ev}} L^2$$

$$c_{\mathrm{ee}} = \beta \gamma_{\mathrm{ee}} L^2$$

$$c_{\mathrm{center}} = \beta g_{\mathrm{center}} L^2$$

$$c_{\mathrm{dangling}} = \beta \gamma_{\mathrm{dangling}} L^2$$

The Typst parameter names are `beta` for β, `gamma-ev` for γ_{ev}, `gamma-ee` for γ_{ee}, `g-center` for g_{center}, and `gamma-dangling` for $\gamma_{\mathrm{dangling}}$. The spring stiffness k is `k-spring`, and the softening constant ε is `eps`.

In `layout-algo: "anneal"`, linnest minimizes an energy:

$$E = \sum_{i<j} \frac{1}{2} \frac{c_{\mathrm{vv}}}{d(v_i, v_j) + \varepsilon} + \sum_{i,e} \frac{c_{\mathrm{ev}}}{d(v_i, e) + \varepsilon} + \sum_{(v,e)\ \mathrm{incident}} \frac{1}{2} k (\ell_e - d(v, e))^2$$

$$+ \sum_{\mathrm{local\ edge\ pairs}} \frac{1}{2} \frac{c_{\mathrm{ee}}}{d(e_i, e_j) + \varepsilon} + \sum_{\mathrm{dangling\ pairs}} \frac{1}{2} \frac{c_{\mathrm{dangling}}}{d(e_i, e_j) + \varepsilon} + \sum_i \frac{1}{2} c_{\mathrm{center}} d(v_i, 0)^2 + p_{\mathrm{cross}} N_{\mathrm{cross}}$$

.

Here p_{cross} is `crossing-penalty` and N_{cross} is the number of detected edge crossings. The quadratic center term pulls nodes toward the origin at every radius. `temp`, `step`, `seed`, `steps`, `epochs`, `cool`, `accept-floor`, `step-shrink`, and `incremental-energy` belong to this simulated annealing mode. `crossing-penalty` is also anneal-only; force mode does not currently add a crossing force.

In `layout-algo: "force"`, linnest applies the direct forces corresponding to the same vertex-vertex, edge-vertex, incidence spring, local edge-edge, dangling-edge, and center terms. `step` is the integration step, `delta` clamps per-step movement, `steps` and `epochs` set the iteration budget, `cool` shrinks the step after each epoch, and `early-tol` stops when movement is small. `z-spring` and `z-spring-growth` are force-only helpers: the integrator gives points temporary z coordinates to break overlaps and pulls them back toward the 2D plane.

`directional-force` is applied in both modes as an extra bias derived from pin/port direction constraints.

After either graph layout mode, labels are relaxed separately. If L_l is the label target distance and q_l is the label repulsion strength, then $L_l = \alpha_l L$ and $q_l = \beta_l L^2$. The Typst names are `label-length-scale` for α_l and `label-charge` for β_l. `label-spring` is the spring constant pulling each label toward its target. `label-layout: "normal"` uses a perpendicular offset target. With `label-layout: "dangling-tangent"`, paired edges still use that perpendicular target, but dangling half-edge labels are offset along the edge direction away from the attached node. With `label-layout: "fixed-length"`, the label remains at distance L_l from the edge point and only rotates around it under repulsive forces. `label-steps`, `label-step`, `label-early-tol`, and `label-max-delta-scale` control the label relaxation iteration.

13.2 Qualified aliases

The exported `layouts` module exposes the same implementations for callers that prefer qualified access; these aliases link back to the canonical top-level entries below.

13.2.1 layouts.layout

Use `layout` for the shared signature and parameter reference.

13.2.2 layouts.sequence

Use `layout-sequence` for the shared multi-pass reference.

13.3 Reference

- layout()
- layout-sequence()

13.3.1 layout

Apply the linnest layout pass to a graph object.

This is intentionally a second step: construct or parse a graph first, then call `layout`. Set `layout-algo` to `"force"` for deterministic force integration, `"anneal"` for simulated annealing, `"tree"` for a traversal tree placement, `"dot"` for a Graphviz-like layered placement, or `"stable-layered"` for a stable railroad-inspired layered placement.

```
#let g = graph.parse("digraph partial { a -> b;a -> b;a:s -> b:s; b:s -> c:s; c:s ->
d:s; d:s -> a:s }").at(0)
#let south = subgraph.compass(g,"s")
#let gf = layout(layout(g, layout-algo: "force"), layout-algo: "tree", layout-nodes:
"fixed",subgraph:south)
#let ga = layout(g, layout-algo: "force")
#let gt = layout(layout(g, layout-algo: "tree",layout-roots: (2)), layout-algo:
"anneal", layout-nodes: "fixed",gamma-ee:0.1,gamma-ev:.75,beta:5,length-scale:0.4)
#grid(columns: 3, gutter: 2cm, draw(gf), draw(ga), draw(gt))
```

```
#let g = graph.parse("digraph partial { a -> b; b -> c; c -> d; d -> a }").at(0)
#let tree = graph.forests(g).at(0)
#let g = layout(g, layout-algo: "tree", subgraph: tree)
#graph.edges(g).map(edge => edge.pos)
```

13.3.1.1 Parameters

```
layout(
  graph: dictionary,
  subgraph: none bytes,
  viewport-w: float,
  viewport-h: float,
  tree-dx: float,
  tree-dy: float,
  steps: int,
  seed: int,
  step: float,
  step-shrink: float,
  cool: float,
  accept-floor: float,
  early-tol: float,
  temp: float,
  delta: float,
  beta: float,
  k-spring: float,
  g-center: float,
  epochs: int,
  crossing-penalty: float,
  gamma-dangling: float,
  gamma-ee: float,
  directional-force: float,
  label-length-scale: float,
  label-spring: float,
  label-charge: float,
  label-steps: int,
  label-layout: string,
  label-step: float,
  label-early-tol: float,
  label-max-delta-scale: float,
  gamma-ev: float,
  eps: float,
  incremental-energy: bool,
  layout-algo: string,
  layout-nodes: string,
  layout-direction: string,
  rank-align: string,
  layout-roots: array,
  rank-same: array,
  route-edge-weight: float,
  route-exit-weight: float,
  route-label-width-scale: float,
  route-label-width-cap: float,
  z-spring: float,
  z-spring-growth: float,
  length-scale: float
) -> dictionary
```

13.3.1.1.1 graph `dictionary`

Graph object returned by `graph.build` or `graph.parse`.

13.3.1.1.2 subgraph `none` or `bytes`

Optional subgraph object to lay out. With `"tree"`, other edges are drawn from the resulting node positions. With `"dot"` and `"stable-layered"`, the subgraph determines rank constraints, while all paired edges between included nodes get dummy routing vertices and edge positions. With `"force"` and `"anneal"`, nodes and edges outside the subgraph are fixed boundary points during optimization.

Default: `none`

13.3.1.1.3 viewport-w `float`

Width of the layout viewport used to derive the natural spring length. Applies to both `"force"` and `"anneal"`.

Default: `10.0`

13.3.1.1.4 viewport-h `float`

Height of the layout viewport used to derive the natural spring length. Applies to both `"force"` and `"anneal"`.

Default: `10.0`

13.3.1.1.5 tree-dx `float`

Horizontal spacing multiplier for traversal-tree and layered placement. For `"force"` and `"anneal"`, this scales the initial placement.

Default: `0.9`

13.3.1.1.6 tree-dy `float`

Vertical spacing multiplier for traversal-tree and layered placement. For `"force"` and `"anneal"`, this scales the initial placement.

Default: `1.2`

13.3.1.1.7 steps `int`

Iterations per epoch. In `"force"` mode this is the number of force integration steps; in `"anneal"` mode this is the number of proposals per temperature epoch.

Default: `int(sys.inputs.at("steps", default: "30"))`

13.3.1.1.8 seed `int`

Seed for deterministic initialization, force-mode jitter, and annealing proposals. Applies to both modes.

Default: `int(sys.inputs.at("seed", default: "2"))`

13.3.1.1.9 step `float`

Initial movement scale. `"force"` multiplies computed forces by this value; `"anneal"` uses it as a proposal step size in natural spring-length units.

Default: `0.81`

13.3.1.1.10 step-shrink `float`

Anneal-only step shrink factor, applied when an epoch's acceptance ratio falls below `accept-floor`.

Default: `0.21`

13.3.1.1.11 cool `float`

Cooling factor applied once per epoch. `"anneal"` cools `temp`; `"force"` shrinks `step`.

Default: `0.85`

13.3.1.1.12 accept-floor `float`

Anneal-only acceptance-ratio threshold below which `step` is shrunk by `step-shrink`.

Default: `0.15`

13.3.1.1.13 early-tol `float`

Force-only early stop threshold for maximum movement in one step, as a multiple of the natural spring length. The annealing schedule stores this value but does not currently use it for stopping.

Default: `1e-6`

13.3.1.1.14 temp `float`

Anneal-only initial temperature used in the Metropolis acceptance test, scaled by natural spring length squared.

Default: `0.3`

13.3.1.1.15 delta `float`

Force-mode maximum movement clamp per point and per step, as a multiple of the natural spring length.

Default: `0.4`

13.3.1.1.16 beta `float`

Base repulsion strength for vertex-vertex interactions. Also scales `gamma-ev`, `gamma-ee`, `gamma-dangling`, and `g-center`. Applies to both modes through the shared spring energy.

Default: `50.0`

13.3.1.1.17 k-spring `float`

Spring stiffness for node-to-edge incidence lengths. Applies to both modes.

Default: `11.0`

13.3.1.1.18 g-center `float`

Centering strength relative to `beta`. Applies to both modes.

Default: `0.002`

13.3.1.1.19 epochs `int`

Number of epochs. Both modes run up to `steps` iterations inside each epoch.

Default: `30`

13.3.1.1.20 crossing-penalty `float`

Anneal-only fixed energy penalty per detected edge crossing. The direct force integrator does not currently add a crossing force.

Default: `30.0`

13.3.1.1.21 gamma-dangling `float`

Repulsion for dangling half edges, relative to `beta`. Applies to both modes through the shared spring energy.

Default: `5.0`

13.3.1.1.22 gamma-ee `float`

Local edge-edge repulsion, relative to `beta`. Applies to both modes.

Default: `0.1`

13.3.1.1.23 directional-force `float`

Bias that pushes points in directions implied by pin/port constraints. Force mode treats this as a force and scales it by natural spring length; anneal mode treats it as a dimensionless multiplier on proposal steps. Applies to both modes.

Default: `5.0`

13.3.1.1.24 label-length-scale `float`

Edge-label target offset as a multiple of the graph spring length. Label layout runs after both graph layout modes.

Default: `0.6`

13.3.1.1.25 label-spring `float`

Spring strength pulling each label toward its target offset in the spring-based label layouts.

Default: `23.0`

13.3.1.1.26 label-charge `float`

Repulsion strength between labels and graph points, scaled by spring length squared. Label layout runs after both modes.

Default: `3.0`

13.3.1.1.27 label-steps `int`

Maximum number of post-layout label relaxation steps. Set to `0` to skip label placement. Applies after both modes.

Default: `20`

13.3.1.1.28 label-layout `string`

Edge-label relaxation model. `"normal"` uses a perpendicular offset, `"dangling-tangent"` uses the edge direction for dangling half-edge labels and a perpendicular offset for paired edges, and `"fixed-length"` keeps each label at a fixed distance from its edge point and only lets that segment rotate.

Default: `"normal"`

13.3.1.1.29 label-step `float`

Label relaxation step size. Applies after both modes.

Default: `0.15`

13.3.1.1.30 label-early-tol `float`

Label relaxation early stop threshold. Applies after both modes.

Default: `1e-3`

13.3.1.1.31 label-max-delta-scale `float`

Label movement clamp as a multiple of spring length. Applies after both modes.

Default: `0.5`

13.3.1.1.32 gamma-ev `float`

Edge-vertex repulsion, relative to `beta`. Applies to both modes.

Default: `0.01`

13.3.1.1.33 eps `float`

Softening epsilon used in inverse-square force/energy terms. Applies to both modes.

Default: `1e-4`

13.3.1.1.34 incremental-energy `bool`

Whether annealing updates cached energy by local deltas. This is anneal-only; force mode computes direct forces instead of energies.

Default: `true`

13.3.1.1.35 layout-algo `string`

Layout algorithm. Use `"force"` for direct force integration, `"anneal"` for simulated annealing against the spring energy, `"tree"` for a traversal-tree placement, `"dot"` for a Graphviz-like layered placement, or `"stable-layered"` for a stable railroad-inspired layered placement.

Default: `"force"`

13.3.1.1.36 layout-nodes `string`

Node movement policy. `"layout"` lets the layout algorithm move nodes. `"fixed"` keeps every node at its current position for this layout pass and only moves edge control points. With `subgraph`, only edges in the subgraph are moved; other edge control points stay at their current positions.

Default: `"layout"`

13.3.1.1.37 layout-direction `string`

Direction for traversal-tree and layered rank placement. `"down"` places increasing ranks downward; `"right"` swaps the layout axes so increasing ranks go left-to-right and measured node/label widths reserve rank-axis space.

Default: `"down"`

13.3.1.1.38 rank-align `string`

Alignment of real nodes inside a rank along the rank axis. `"center"` keeps node centers aligned, while `"start"` / `"left"` and `"end"` / `"right"` align the corresponding measured node-box side.

Default: `"center"`

13.3.1.1.39 layout-roots `array`

Ordered node indices used as preferred roots for `"tree"`, `"dot"`, and `"stable-layered"`. Roots outside the selected node set are ignored. Remaining components are laid out afterward in graph order.

Default: `()`

13.3.1.1.40 rank-same `array`

Subgraphs whose incident nodes should share a dot/stable-layered rank. These are layout hints supplied by Typst rather than parsed graph structure.

Default: `()`

13.3.1.1.41 route-edge-weight `float`

Dot/stable-layered also honors a node statement `layout-rank` as an exact non-negative integer rank. Nodes with the same `layout-rank` are placed on the same horizontal layer, and larger ranks are placed lower. Relative layout weight for paired edges outside the dot/stable-layered rank subgraph. Lower values make these edges guide routing without dominating the rank tree.

Default: `0.15`

13.3.1.1.42 route-exit-weight `float`

Extra horizontal straightening weight for the first or last segment of an edge with `source-route-exit` or `sink-route-exit` set to a vertical side.

Default: `4.0`

13.3.1.1.43 route-label-width-scale `float`

Multiplier for measured edge-label width when sizing non-rank dummy routing vertices in dot/stable-layered layout.

Default: `1.0`

13.3.1.1.44 route-label-width-cap `float`

Maximum non-rank dummy label width as a multiple of `tree-dx`. Set to `0` or a negative value to disable the cap.

Default: `2.0`

13.3.1.1.45 z-spring `float`

Force-only spring pulling temporary z coordinates back toward the layout plane. Higher values keep the visible 2D forces from being hidden by the temporary 3D symmetry-breaking offsets.

Default: `2.0`

13.3.1.1.46 z-spring-growth `float`

Force-only per-epoch multiplier for `z-spring`.

Default: `1.0`

13.3.1.1.47 length-scale `float`

Natural spring-length multiplier. This scales the graph's preferred edge length and dimensional force/energy terms so changing only this value mostly zooms the result instead of retuning the force ratios. Applies to both modes.

Default: `0.35`

13.3.2 layout-sequence

Apply multiple layout passes in order.

Each pass is a dictionary of named arguments accepted by `layout`, excluding the graph itself.

13.3.2.1 Parameters

```
layout-sequence(
  graph: dictionary,
  passes: array
) -> dictionary
```

13.3.2.1.1 graph `dictionary`

Graph object returned by `graph.build` or `graph.parse`.

13.3.2.1.2 passes `array`

Array of layout option dictionaries.

14 Drawing functions

`draw`, `edge-halves`, and `to-cetz-edge-halves` are top-level functions exported by `lib.typ`. They turn an already laid-out graph into CeTZ content and expose the edge geometry used by custom renderers.

14.1 Concepts

14.1.1 Drawing

Draw styling is Typst-native. Pass dictionaries or callbacks to `draw`; edge callbacks receive the merged `scope`, edge statements, source/sink half-edge records, and the edge index:

```
#let edge-label(edge) = text(fill: rgb("#" + edge.color))[#edge.display-label]
#let source-style(edge) = (stroke: red + 0.5pt)
#let sink-style(edge) = (stroke: blue + 0.5pt)
#draw(layout(g), edge-label: edge-label, source-style: source-style, sink-style: sink-
style)
```

Edge labels are drawn with CeTZ content at the layout label position. Use `edge-label-style: (anchor: "south")`, or an edge-data callback returning a style dictionary, to choose which point of the label is anchored there.

Marks can follow graph orientation as a first-class draw option. Put the same mark layer on both halves and set `mark-orientation: "edge"`; default-oriented edges mark the source half, reversed edges mark the sink half with the marker flipped, and undirected edges suppress the mark.

```
#let oriented-arrow = (
  stroke: black + 0.7pt,
  mark: (end: (symbol: ">", fill: black, anchor: "center", shorten-to: auto), scale: 0.75),
  mark-position: "center-if-dangling",
  mark-orientation: "edge",
)
#draw(layout(g), source-style: oriented-arrow, sink-style: oriented-arrow)
```

14.2 Reference

- edge-halves()
- to-cetz-edge-halves()
- draw()

14.2.1 edge-halves

Split a laid-out graph edge into source and sink half-edge paths.

The returned dictionary has `source`, `sink`, and `curve`. The split point is the edge layout point, so the two half-edges join smoothly there.

14.2.1.1 Parameters

```
edge-halves(
  edge: dictionary,
  nodes: array,
  omega: float,
  source-outset: int float,
  sink-outset: int float,
  accuracy: float
) -> dictionary
```

14.2.1.1.1 edge `dictionary`

Edge record returned by `graph.edges(layout(g))`.

14.2.1.1.2 nodes `array`

Node records returned by `graph.nodes(layout(g))`.

14.2.1.1.3 omega `float`

Hobby curl used for the source-to-edge and edge-to-sink curves.

Default: `1.0`

14.2.1.1.4 source-outset `int` or `float`

Arc-length trim applied at the source node side.

Default: `0`

14.2.1.1.5 sink-outset `int` or `float`

Arc-length trim applied at the sink node side.

Default: `0`

14.2.1.1.6 accuracy `float`

Arc-length accuracy used while trimming.

Default: `0.001`

14.2.2 to-cetz-edge-halves

Draw the two halves of a laid-out graph edge through CeTZ.

`source-style` applies from the source node to the edge layout point, and `sink-style` applies from the edge layout point to the sink node.

14.2.2.1 Parameters

```
to-cetz-edge-halves(
  edge: dictionary,
  nodes: array,
  unit: int float length ratio,
  omega: float,
  source-outset: int float,
  sink-outset: int float,
  accuracy: float,
  source-style: dictionary,
  sink-style: dictionary
) -> content
```

14.2.2.1.1 edge `dictionary`

Edge record returned by `graph.edges(layout(g))`.

14.2.2.1.2 nodes `array`

Node records returned by `graph.nodes(layout(g))`.

14.2.2.1.3 unit `int` or `float` or `length` or `ratio`

Coordinate length for one graph-layout unit. Numbers are interpreted as em.

Default: `1`

14.2.2.1.4 omega `float`

Hobby curl used for the source-to-edge and edge-to-sink curves.

Default: `1.0`

14.2.2.1.5 source-outset `int` or `float`

Arc-length trim applied at the source node side.

Default: `0`

14.2.2.1.6 sink-outset `int` or `float`

Arc-length trim applied at the sink node side.

Default: `0`

14.2.2.1.7 accuracy `float`

Arc-length accuracy used while trimming.

Default: `0.001`

14.2.2.1.8 source-style `dictionary`

CeTZ style for the source half edge.

Default: `(:)`

14.2.2.1.9 sink-style `dictionary`

CeTZ style for the sink half edge.

Default: `(:)`

14.2.3 draw

Draw a graph object with CeTZ.

The graph must already have positions, either from `layout` or from explicit `pos` values passed to graph node or edge items. Paired edges without an explicit edge position use the midpoint of their endpoint nodes. Node and edge Typst style dictionaries or callbacks are evaluated and forwarded to CeTZ.

```
 #let positioned = graph.build({
   graph.node(<left>, label: [left], pos: graph.pos(x: 0, y: 0))
   graph.node(<right>, label: [right], pos: graph.pos(ref: <left>, dx: 2.5, dy: 0))
   graph.edge(graph.source(<left>), graph.sink(<right>))
   graph.edge(graph.source(<right>), <right-out>, pos: graph.pos(ref: <right>, dx: 0.9, dy:
0.7))
 },
   name: "positioned",
 )
 #draw(positioned, source-style: (stroke: black + 0.7pt), sink-style: (stroke: black +
0.7pt))
```

```
 #let parallel-base-edge = 0
 #let source-patterns = (
   "coil",
   "zigzag",
   "coil",
   "wave",
   "zigzag",
   "coil",
   "wave",
   "zigzag",
 )
```

```
#let sink-patterns = (
  "coil",
  "zigzag",
  "coil",
  "zigzag",
  "coil",
  "wave",
  "coil",
  "wave",
)
#let parallel-layer(edge, mark: none) = if edge.eid == parallel-base-edge {
  (
    offset: -0.5,
    length: 1.5,
    ratio: 0.5,
    resolve-length: "min",
    stroke: (paint: rgb("#2f6f4e"), thickness: 1.1pt, cap: "round"),
    mark: mark,
  )
} else { none }
#let stack(base, layer) = if layer == none { base } else { (base, layer) }
#let source-stroke(edge) = if edge.eid == parallel-base-edge {
  (paint: gray, thickness: 0.75pt, cap: "round")
} else {
  (paint: red, thickness: 1.2pt, cap: "round")
}
#let sink-stroke(edge) = if edge.eid == parallel-base-edge {
  (paint: gray, thickness: 0.75pt, cap: "round")
} else {
  (paint: blue, thickness: 1.2pt, cap: "round", dash: "dotted")
}
#let source-style(edge) = {
  let base = (
    stroke: source-stroke(edge),
    pattern: source-patterns.at(edge.eid),
    pattern-amplitude: 0.18,
    pattern-wavelength: 0.55,
    pattern-coil-longitudinal-scale: 1.6,
  )
  stack(base, parallel-layer(edge))
}
#let sink-style(edge) = {
  let base = (
    stroke: sink-stroke(edge),
    pattern: sink-patterns.at(edge.eid),
    pattern-amplitude: 0.18,
    pattern-wavelength: 0.55,
    pattern-coil-longitudinal-scale: 1.6,
  )
  stack(base, parallel-layer(edge, mark: (end: (symbol: "straight"), scale: 0.75)))
}
#let g = graph.build({
  graph.node(<a>, label: [a hi])
  graph.node(<c>)
  graph.node(<d>)
  graph.node(<e>)
  graph.edge(graph.source(<a>), <ac>, graph.sink(<c>), pos: graph.pos(x: 0, y: 0.75, mode:
"pin"))
  graph.edge(graph.source(<c>), <ca>, graph.sink(<a>, compass: "e"))
  graph.edge(graph.source(<c>), <cd>, graph.sink(<d>, compass: "e"))
```

```
  graph.edge(graph.source(<e>), <ed>, graph.sink(<d>, compass: "e"))
  graph.edge(graph.source(<e>), <ea>, graph.sink(<a>, compass: "e"))
  graph.edge(graph.source(<d>), <d-out>)
  graph.edge(graph.source(<e>, compass: "e"), <e-out>)
  graph.edge(graph.source(<a>), <a-out>)
},
  name: "demo",
)
#let layed-out = layout(g)
#let east = subgraph.compass(layed-out, "e")
#draw(layed-out, subgraph: east, source-style: source-style, sink-style: sink-style)
```

```
#let p = graph.build({
  graph.node(<pa>, label: [a], pos: graph.pos(y: 0, mode: "pin"))
  graph.node(<pc>, label: [c], pos: graph.pos(y: 0, mode: "pin"))
  graph.node(<pc1>, label: [c], pos: graph.pos(y: 0, mode: "pin"))
  graph.node(<pc2>, label: [c], pos: graph.pos(y: 0, mode: "pin"))
  graph.edge(graph.source(<pa>), <e0>, graph.sink(<pc>))
  graph.edge(graph.source(<pc2>), <e1>, graph.sink(<pc1>))
},
  name: "parallel demo",
)
#let parallel-edge-style(edge) = (
    offset: -0.5,
    length: 1.4,
    ratio: 0.5,
    resolve-length: "min",
    stroke: (paint: rgb("#2f6f4e"), thickness: 1.1pt, cap: "round"),
  )

#let focused-base-style(edge) = (
  stroke: (paint: gray, thickness: 0.7pt, cap: "round"),
  pattern: "coil",
  pattern-amplitude: 0.14,
  pattern-wavelength: 0.45,
  pattern-coil-longitudinal-scale: 1.5,
)
#let focused-source-style(edge) = (focused-base-style(edge), parallel-edge-style(edge))
#let focused-sink-style(edge) = (focused-base-style(edge), parallel-edge-style(edge) + (
  mark: (end: ">"),
))
#draw(layout(p, g-center: 0.004, label-steps: 0,), unit: 1.25, source-style: focused-
source-style, sink-style: focused-sink-style)
```

```
#let g = graph.build({
  graph.node(<a>, pos: graph.pos(x: 0, y: 0, mode: "pin"))
  graph.node(<c>, pos: graph.pos(x: 3, y: 0, mode: "pin"))
  graph.node(<d>, pos: graph.pos(x: 3, y: -1, mode: "pin"))
  graph.edge(graph.source(<a>), <ac>, graph.sink(<c>), orientation: "default")
  graph.edge(graph.source(<a>), <ad>, graph.sink(<d>), orientation: "reversed")
},
  name: "oriented marks",
)
#let arrow = (
```

```
  end: (symbol: ">", fill: black, anchor: "center", shorten-to: auto),
  scale: 0.75,
)
#let oriented-arrow = (
  stroke: black + 0.7pt,
  mark: arrow,
  mark-position: "center-if-dangling",
  mark-orientation: "edge",
)
#draw(layout(g), unit: 1.4, source-style: oriented-arrow, sink-style: oriented-arrow)
```

14.2.3.1 Parameters

```
draw(
  graph: bytes,
  scope: dictionary,
  unit: auto int float length ratio,
  title: none auto content string,
  subgraph: none bytes array,
  debug: bool int,
  node-radius: auto int float array,
  node-min-radius: int float,
  node-label-padding: int float,
  node-fill: any,
  node-stroke: any,
  node-outset: auto int float,
  node-label-style: dictionary,
  node-style: dictionary function none,
  node-label: auto content string function none,
  draw-node: auto function,
  edge-stroke: any,
  edge-offset: int float,
  edge-length: none int float,
  edge-ratio: none int float,
  edge-resolve-length: string function,
  edge-accuracy: float,
  edge-optimize: bool,
  source-style: dictionary array function none,
  sink-style: dictionary array function none,
  edge-label: content string function none,
  edge-label-style: dictionary function none,
  edge-omega: float,
  edge-trim-accuracy: float,
  padding: none int float array dictionary,
  debug-edge-radius: int float,
  debug-edge-fill: any,
  debug-edge-stroke: any,
  debug-edge-label-fill: any,
  subgraph-edge-style: dictionary,
  subgraph-edge-underlay: bool
) -> content
```

14.2.3.1.1 graph `bytes`

Graph object with positions from `layout` or explicit graph API `pos` fields.

14.2.3.1.2 scope `dictionary`

Additional values merged into node and edge callback dictionaries.

Default: `(:)`

14.2.3.1.3 unit `auto` or `int` or `float` or `length` or `ratio`

Coordinate length for one graph-layout unit. `auto` uses any unit stored by `graph.style`, falling back to `1em`. Numbers are interpreted as em.

Default: `auto`

14.2.3.1.4 title `none` or `auto` or `content` or `string`

Optional title displayed above the diagram. Use `auto` for the graph name.

Default: `none`

14.2.3.1.5 subgraph `none` or `bytes` or `array`

Optional subgraph or array of subgraphs whose half-edges are shaded. Array entries may be raw subgraphs or records like `(subgraph: sg, edge-style: (stroke: red + 2pt))`.

Default: `none`

14.2.3.1.6 debug `bool` or `int`

Debug level. `1` enables CeTZ canvas debug; `2` also marks edge positions.

Default: `false`

14.2.3.1.7 node-radius `auto` or `int` or `float` or `array`

Default CeTZ node radius. Use `auto` to fit the node label.

Default: `auto`

14.2.3.1.8 node-min-radius `int` or `float`

Minimum radius used when `node-radius` is `auto`.

Default: `0.16`

14.2.3.1.9 node-label-padding `int` or `float`

Extra canvas-unit padding around labels when `node-radius` is `auto`.

Default: `0.08`

14.2.3.1.10 node-fill `any`

Default CeTZ node fill.

Default: `white`

14.2.3.1.11 node-stroke `any`

Default CeTZ node stroke.

Default: `black`

14.2.3.1.12 node-outset `auto` or `int` or `float`

Edge clearance from node centers. `auto` uses each node circle radius. Increase this when labels extend beyond the circle.

Default: `auto`

14.2.3.1.13 node-label-style `dictionary`

Default node-label style forwarded to `cetz.draw.content`.

Default: `(:)`

14.2.3.1.14 node-style `dictionary` or `function` or `none`

Node style dictionary or callback. A callback receives node data.

Default: `(:)`

14.2.3.1.15 node-label `auto` or `content` or `string` or `function` or `none`

Node label content or callback. `auto` uses the node name.

Default: `auto`

14.2.3.1.16 draw-node `auto` or `function`

CeTZ-compatible node drawing callback. `auto` draws the default circular node and label. A callback receives `(node, box)` and should return CeTZ draw elements; `box` contains `name`, `center`, `width`, `height`, `unit`, `label`, `label-style`, `style`, `radius`, and `node`.

Default: `auto`

14.2.3.1.17 edge-stroke `any`

Default CeTZ edge stroke.

Default: `0.1em`

14.2.3.1.18 edge-offset `int` or `float`

Default normal offset for edge paths. Applied to the base edge geometry before patterns; node outsets then trim the shifted path.

Default: `0`

14.2.3.1.19 edge-length `none` or `int` or `float`

Maximum visible arc length for centered parallel edge paths. `none` keeps the full shifted path.

Default: `none`

14.2.3.1.20 edge-ratio `none` or `int` or `float`

Maximum visible fraction of the base edge length for centered parallel edge paths. Combined with `edge-length` according to `edge-resolve-length`.

Default: `none`

14.2.3.1.21 edge-resolve-length `string` or `function`

Resolve `edge-length` and `edge-ratio`. Accepted string values are `"min"`/`"shorter"`, `"max"`/`"longer"`, `"length"`/`"fixed"`, `"ratio"`/`"relative"`, or `"none"`/`"full"`. A function receives `(base-length, length, ratio)`.

Default: `"min"`

14.2.3.1.22 edge-accuracy `float`

Arc-length accuracy for fitted parallel edge paths.

Default: `0.001`

14.2.3.1.23 edge-optimize `bool`

Let Kurbo optimize the fitted parallel path.

Default: `true`

14.2.3.1.24 source-style `dictionary` or `array` or `function` or `none`

Source half-edge style dictionary, array of layer dictionaries, or callback. `mark-position: "center-if-dangling"` keeps an end marker at the paired-edge split point while centering it on dangling half edges. `mark-orientation: "edge"` makes a mark follow `edge.orientation` instead of raw path direction; reversed edges move the mark to the sink half and flip it, while undirected edges suppress it. `source-anchor` may be a CeTZ anchor name such as `"north"` or `"south"` to route this endpoint from a measured node-box anchor. By default, anchored paired edges use two smooth cubic halves through the edge layout point while preserving the endpoint anchor tangents. Set `route: "hobby-through"` to instead build a single Hobby spline through the source anchor, source guide, edge point, sink guide, and sink anchor, then split source/sink styling at the edge point. `route: "direct"` keeps the same anchored cubic routing but suppresses the default edge-position Hobby route. `route-points: "through"` also threads any layout-provided half-edge route points through that same Hobby path. `route: "straight-through"` draws the two straight force springs from source to edge position and from edge position to sink.

Default: `(:)`

14.2.3.1.25 sink-style `dictionary` or `array` or `function` or `none`

Sink half-edge style dictionary, array of layer dictionaries, or callback. A callback receives edge data. `mark-position: "center-if-dangling"` has the same dangling-edge behavior as for `source-style`, and `mark-orientation: "edge"` participates in the same orientation-aware mark placement. `sink-anchor` may be a CeTZ anchor name such as `"north"` or `"south"` to route this endpoint into a measured node-box anchor. `route: "direct"` has the same meaning as in `source-style`.

Default: `(:)`

14.2.3.1.26 edge-label `content` or `string` or `function` or `none`

Edge label content or callback. A callback receives edge data.

Default: `none`

14.2.3.1.27 edge-label-style `dictionary` or `function` or `none`

Edge-label style forwarded to `cetz.draw.content`; use `anchor` to choose which point of the label is placed at the layout label position. A callback receives edge data.

Default: `(:)`

14.2.3.1.28 edge-omega `float`

Hobby curl used at the endpoints of paired edge curves.

Default: `1.0`

14.2.3.1.29 edge-trim-accuracy `float`

Optional style key for anchored source/sink routes. Set `anchor-control-distance` in `source-style` or `sink-style` to override the automatic guide distance used by cubic anchored routes. -> auto | int | float Arc-length accuracy for trimming edge curves at node outsets.

Default: `0.001`

14.2.3.1.30 padding `none` or `int` or `float` or `array` or `dictionary`

CeTZ canvas padding.

Default: `0.4`

14.2.3.1.31 debug-edge-radius `int` or `float`

Radius for edge-position markers shown at `debug >= 2`.

Default: `0.08`

14.2.3.1.32 debug-edge-fill `any`

Fill for edge-position markers shown at `debug >= 2`.

Default: `rgb("#ff9f1c")`

14.2.3.1.33 debug-edge-stroke `any`

Stroke for edge-position markers shown at `debug >= 2`.

Default: `rgb("#d72638") + 0.35pt`

14.2.3.1.34 debug-edge-label-fill `any`

Label fill for edge-position markers shown at `debug >= 2`.

Default: `rgb("#7a1020")`

14.2.3.1.35 subgraph-edge-style `dictionary`

Default CeTZ style used to shade half-edges included in `subgraph`. Individual subgraph records may override this with `edge-style`.

Default: `(stroke: rgb("#ffd166") + 4.5pt)`

14.2.3.1.36 subgraph-edge-underlay `bool`

Draw subgraph shading below the normal half-edge style.

Default: `true`

15 physics module

The `physics` module provides reusable source/sink strokes, particle-line patterns, fermion-flow marks, momentum arrows, and label callbacks for `draw`.

15.1 Concepts

15.1.1 Physics Edge Styles

`physics.style(..)` returns `source-style`, `sink-style`, and `edge-label` callbacks for `draw`. The default source/sink strokes use darker/lighter halves to encode the graph source/sink split. Fermion map entries marked with `fermion-flow` receive one particle-flow arrow on the main edge using `mark-orientation: "edge"`; paired edges follow `edge.orientation`, dangling half edges place the arrow in the middle of the visible half edge, and undirected edges omit the arrow. This is independent of momentum arrows.

Set `momentum-arrows: true` to draw centered parallel arrow decorations while the main edge remains connected to the nodes. The arrows flow from source to sink independently of the DOT `dir`/`orientation` value; there is one momentum marker per edge, even though the base edge is internally styled as source and sink halves. On paired edges the marker lives on the sink half; on dangling edges it lives on the existing source or sink half-edge. The decoration defaults to a plain black `0.55pt` stroke and a scaled CeTZ `"barbed"` arrowhead, so it does not inherit the base particle stroke. The arrow length is capped by both `momentum-arrow-length` and `momentum-arrow-ratio`, so the shorter one wins. When the layout provides an edge-label position, the arrow offset is signed so the momentum marker is drawn on the same side of the edge as that label. Use `momentum-arrow-offset` to set the normal displacement. `momentum-arrow-mark: auto` uses the default barbed marker, `momentum-arrow-mark: none` draws only the momentum line, and any other value overrides the CeTZ mark style, for example `"stealth"` or `(end: "barbed", scale: 1.6)`. Only the `end` mark is kept so source/sink splitting cannot create a second momentum head.

Optional labels can be built from edge metadata with `show-edge-index`, `show-half-edge-index`, `show-particle`, and, for explicit momentum fields, `show-momentum`. `show-half-edge-index` only emits a value for dangling half edges.

```
#import "../src/lib.typ": draw, graph, layout, physics
#import graph: build, dot, edge, edges, node, nodes, parse, sink, source

#let g = build({
  node(<a>)
  node(<c>)
  edge(
    source(<a>),
    <fermion-main>,
    sink(<c>),
    id: 7,
    particle: "fermion",
  )
  edge(
    source(<c>),
    <fermion-out>,
    id: 8,
```

```
    particle: "fermion",
  )
},
  name: "physics",
)
#let callbacks = physics.style(
  momentum-arrows: true,
  show-edge-index: true,
  show-particle: true,
)
#draw(
  layout(g),
  source-style: callbacks.source-style,
  sink-style: callbacks.sink-style,
  edge-label: callbacks.edge-label,
)
```

Set `edge-offset` on `draw`, or `offset` in a `source-style`/`sink-style` dictionary, to draw a fitted parallel path. Style callbacks may also return an array of dictionaries; the layers are drawn in order on the same graph edge, so parallel strokes do not require duplicate graph edges.

```
#let base-style(edge) = (
  stroke: (paint: gray, thickness: 0.7pt, cap: "round"),
)
#let offset-style(edge) = (
  offset: 0.18,
  length: 1.4,
  ratio: 0.5,
  resolve-length: "min",
  stroke: (paint: rgb("#2f6f4e"), thickness: 1pt, cap: "round"),
)
#let source-style(edge) = (base-style(edge), offset-style(edge))
#let sink-style(edge) = (base-style(edge), offset-style(edge) + (mark: (end: ">")))
```

The parallel path is applied to the base edge geometry before patterns and other decorations; node outsets then trim the shifted path, so shifted paths still start and end outside fitted node circles. Add `edge-length` or `length` to center-trim the shifted path to a fixed arc length, and add `edge-ratio` or `ratio` to cap it by a fraction of the base edge length. `edge-resolve-length`
`resolve-length` decides how to combine both limits `"min"`/`"shorter"` (default), `"max"`/`"longer"`,

`"length"`/`"fixed"`, `"ratio"`/`"relative"`, `"none"`/`"full"`, or a function receiving `(base-length, length, ratio)`. Set `offset-side: "label"` on an offset layer to choose the sign of `offset` so the layer is drawn on the same side of the curve as the edge label.

Data defaults are also the global styling hook for all sources, sinks, nodes, and edges. More specific data can be added with captured named arguments on node, edge, source, and sink items, or by running `graph.map`/`graph.eval-fields` after construction. This keeps evaluated Typst values in the native data channel instead of adding renderer-specific eval fields to the Rust topology spec:

```
#let g = build({
  node(<a>)
  node(<c>)
  edge(
    source(<a>),
    <a-c>,
    sink(<c>),
    label: [a-c],
    kind: "highlight",
```

```
  )
},
  name: "demo",
  default-source-data: (style: (stroke: red + 0.5pt)),
  default-sink-data: (style: (stroke: blue + 0.5pt)),
)
#let callbacks = physics.style()
#draw(layout(g), source-style: callbacks.source-style, sink-style: callbacks.sink-style)
```

`graph.build` also accepts comma-separated items:

```
#let g = build(
  node(<a>),
  node(<b>),
  edge(
    source(<a>, compass: "e"),
    <ab>,
    sink(<b>, compass: "w"),
    label: [ab],
    statements: (color: "0055ff", label: "ab"),
  ),
  name: "demo",
  statements: (full_num: "x + y"),
  default-node-statements: (shape: "circle"),
  default-edge-statements: (
    color: "000000",
    display-label: "$#label$",
  ),
)
```

15.2 physics

- stroke-style()
- source-stroke()
- sink-stroke()
- particle-name()
- edge-entry()
- text-value()
- eval-expression()
- label-content()
- style-dict()
- momentum-value()
- edge-index()
- dangling-half-edge-index()
- source-style()
- sink-style()
- edge-label()
- style()

Variables

- massive
- massless
- dashed
- dotted
- fermion-arrow-mark
- fermion-flow
- wave

- coil
- zigzag
- default-edge
- default-map
- momentum-arrow-defaults

15.2.1 stroke-style

Build a rounded CeTZ stroke dictionary accepted by `linnest.draw`.

15.2.1.1 Parameters

```
stroke-style(
  c,
  thickness,
  dash
) -> dictionary
```

15.2.2 source-stroke

Source-half stroke helper.

15.2.2.1 Parameters

```
source-stroke(
  c,
  thickness,
  dash
) -> dictionary
```

15.2.3 sink-stroke

Sink-half stroke helper. The default lightening makes the two halves encode the graph's source/sink split without requiring arrowheads.

15.2.3.1 Parameters

```
sink-stroke(
  c,
  thickness,
  dash,
  lighten
) -> dictionary
```

15.2.4 particle-name

Return an edge's particle name, stripping the quotes often present in DOT statement values.

15.2.4.1 Parameters

```
particle-name(edge) -> none string
```

15.2.5 edge-entry

Return the style-map entry for an edge.

15.2.5.1 Parameters

```
edge-entry(
  edge,
  map,
  default
) -> dictionary
```

15.2.6 text-value

Convert DOT-ish values to plain text.

15.2.6.1 Parameters

```
text-value(value) -> string
```

15.2.7 eval-expression

Evaluate a string-valued style expression in the edge scope.

15.2.7.1 Parameters

```
eval-expression(
  value,
  edge,
  map,
  scope
) -> any
```

15.2.8 label-content

Convert a style label value to content. In `mode: "eval"`, string values are evaluated in the edge scope.

15.2.8.1 Parameters

```
label-content(
  value,
  edge,
  mode,
  map,
  scope
) -> none content
```

15.2.9 style-dict

Convert a style value to a dictionary. In `mode: "eval"`, string values are evaluated in the edge scope.

15.2.9.1 Parameters

```
style-dict(
  value,
  edge,
  mode,
  map,
  scope
) -> dictionary
```

15.2.10 momentum-value

Return the momentum field used by optional edge labels.

15.2.10.1 Parameters

```
momentum-value(
  edge,
  fields
) -> none any
```

15.2.11 edge-index

Return the edge index used by optional edge labels. The DOT `id` statement wins over the renderer-local `eid`.

15.2.11.1 Parameters

```
edge-index(
  edge,
  fields
) -> none any
```

15.2.12 dangling-half-edge-index

Return the exposed half-edge index for dangling edges only.

15.2.12.1 Parameters

```
dangling-half-edge-index(edge) -> none any
```

15.2.13 source-style

Style callback for the source half edge.

`momentum-arrows: true` adds one centered black CeTZ-mark decoration while the main edge remains drawn with its normal particle style.

15.2.13.1 Parameters

```
source-style(
  edge,
  map,
  default,
  typst-fields,
  scope,
  orientation-split,
  momentum-arrows,
  momentum-arrow-offset,
  momentum-arrow-length,
  momentum-arrow-ratio,
  momentum-arrow-stroke,
  momentum-arrow-mark
) -> dictionary array
```

15.2.14 sink-style

Style callback for the sink half edge.

`momentum-arrows: true` adds one centered black CeTZ-mark decoration toward the sink node, so momentum arrows always flow from source to sink independently of `edge.orientation`.

15.2.14.1 Parameters

```
sink-style(
  edge,
  map,
  default,
  typst-fields,
  scope,
  orientation-split,
  momentum-arrows,
  momentum-arrow-offset,
  momentum-arrow-length,
  momentum-arrow-ratio,
  momentum-arrow-stroke,
  momentum-arrow-mark
) -> dictionary array
```

15.2.15 edge-label

Edge-label callback.

By default this preserves data `display-label` or `label`, then explicit `display-label`, `label`, and particle-map labels. Set any `show-*` option to build a label from selected metadata instead:

```
#let callbacks = physics.style(
  momentum-arrows: true,
  show-edge-index: true,
  show-particle: true,
)
```

15.2.15.1 Parameters

```
edge-label(
  edge,
  map,
  default,
  typst-fields,
  scope,
  show-momentum,
  show-edge-index,
  show-half-edge-index,
  show-particle,
  momentum-fields,
  edge-index-fields,
  momentum-prefix,
  edge-index-prefix,
  half-edge-index-prefix,
  particle-prefix,
  label-separator,
  ],
  label-size,
  label-fill
) -> none content
```

15.2.16 style

Bundle source-style, sink-style, and edge-label callbacks with shared options for `linnest.draw`.

```
#let g = build({
  node(<a>,pos:pos(y:0))
  node(<b>,pos:pos(y:0))
  edge(source(<a>), <g-edge>, sink(<b>), particle: "g")
  edge(source(<a>), <g-edge2>, sink(<b>), particle: "g")
  edge(source(<a>), <g-edge3>, sink(<b>), particle: "g")
},
  name: "physics",
)
#let a = ```dot
digraph {
0 [dod="-100" num="1"];
 1 [dod="-100" num="1"];
 2 [dod="-100" num="1"];
 3 [dod="-100" num="1"];
 4 [dod="-100" num="1"];
 5 [dod="-100" num="1"];
 6 [dod="-100" num="1"];
  exte0  [style=invis];
  exte0 -> 2:0   [id=0  dod="-100" lmb_rep="P(0,a___)" num="1" particle="d" pin="x:@-
left"];
  exte1  [style=invis];
  exte1 -> 1:1   [id=1 dir=back  dod="-100" lmb_rep="P(1,a___)" num="1" particle="d~"
pin="x:@-left"];
  exte2  [style=invis];
  exte2 -> 1:2   [id=2 dir=none  dod="-100" lmb_rep="P(2,a___)" mass="0" num="1"
particle="H" pin="x:@-left"];
```

```
  exte3  [style=invis];
  exte3 -> 2:3   [id=3 dir=none  dod="-100" is_dummy="true" lmb_rep="0" mass="0"
num="1" particle="H" pin="x:@-left"];
  exte4  [style=invis];
  6:4 -> exte4   [id=4 dir=none  dod="-100" lmb_rep="P(3,a___)" mass="0" num="1"
particle="H" pin="x:@+right"];
  exte5  [style=invis];
  1:5 -> exte5   [id=5 dir=none  dod="-100" is_dummy="true" lmb_rep="0" mass="0"
num="1" particle="H" pin="x:@+right"];
  exte6  [style=invis];
  5:6 -> exte6   [id=6 dir=none  dod="-100" lmb_rep="P(4,a___)" num="1" particle="H"
pin="x:@+right"];
  exte7  [style=invis];
  4:7 -> exte7   [id=7 dir=none  dod="-100" lmb_rep="P(5,a___)" num="1" particle="H"
pin="x:@+right"];
  exte8  [style=invis];
  3:8 -> exte8   [id=8 dir=none  dod="-100"
lmb_rep="-1*P(3,a___)+-1*P(4,a___)+-1*P(5,a___)+P(0,a___)+P(1,a___)+P(2,a___)"
num="1" particle="H" pin="x:@+right"];
  0:9 -> 1:10  [id=9  dod="-100" lmb_id="1" lmb_rep="K(1,a___)" num="1"
particle="t"];
  0:11  -> 2:12  [id=10 dir=none  dod="-100" lmb_rep="-1*P(0,a___)+K(0,a___)" num="1"
particle="g"];
  5:13  -> 0:14  [id=11  dod="-100" lmb_rep="-1*P(0,a___)+K(0,a___)+K(1,a___)"
num="1" particle="t"];
  6:24  -> 1:18  [id=12 dir=none  dod="-100" lmb_rep="-1*P(3,a___)+K(0,a___)"
mass="((Q(2,spenso::cind(0)))^2+(Q(2,spenso::cind(1)))^2*-1+(Q(2,spenso::cind(2)))^2*-1+(Q(2,spens
num="1" particle="g"];
  1:15  -> 4:16  [id=13  dod="-100"
lmb_rep="-1*P(3,a___)+K(0,a___)+K(1,a___)+P(1,a___)+P(2,a___)" num="1" particle="t"];
  2:17  -> 6:23  [id=14  dod="-100" lmb_id="0" lmb_rep="K(0,a___)" num="1"
particle="d"];
  4:19  -> 3:20  [id=15  dod="-100"
lmb_rep="-1*P(3,a___)+-1*P(5,a___)+K(0,a___)+K(1,a___)+P(1,a___)+P(2,a___)" num="1"
particle="t"];
  3:21  -> 5:22  [id=16  dod="-100"
lmb_rep="-1*P(0,a___)+K(0,a___)+K(1,a___)+P(4,a___)" num="1" particle="t"];
}

 ```
 #let (source-style, sink-style, edge-label, ..callbacks) = physics.style(momentum-
arrows: true, show-edge-index: true)
 #draw(
   layout(g,beta: 100),
   source-style: source-style,
   sink-style: sink-style,
   edge-label: edge-label,
)
 #let g = parse(a.text).at(0)
 #draw(
   layout(layout(g,beta: 100,epochs:300,steps:200),subgraph: subgraph.bits(g,
(false,false,false,false,false,true,true,true,true,true,true,true,true,true,true,true,true,tr
   source-style: source-style,
   sink-style: sink-style,
   edge-label: edge-label,
)
```
```

15.2.16.1 Parameters

```
style(
  map,
  default,
  typst-fields,
  scope,
  orientation-split,
  momentum-arrows,
  momentum-arrow-offset,
  momentum-arrow-length,
  momentum-arrow-ratio,
  momentum-arrow-stroke,
  momentum-arrow-mark,
  show-momentum,
  show-edge-index,
  show-half-edge-index,
  show-particle,
  momentum-fields,
  edge-index-fields,
  momentum-prefix,
  edge-index-prefix,
  half-edge-index-prefix,
  particle-prefix,
  label-separator,
  ],
  label-size,
  label-fill
) -> dictionary
```

15.2.17 massive

`length`

Conventional massive-particle stroke thickness.

15.2.18 massless

`length`

Conventional massless-particle stroke thickness.

15.2.19 dashed

`array`

Dash pattern used for scalar-style lines.

15.2.20 dotted

`string`

Dotted dash style used for ghost-style lines.

15.2.21 fermion-arrow-mark `dictionary`

CeTZ marker used for fermion particle-flow arrows on the main edge.

15.2.22 fermion-flow `dictionary`

Mark an edge-map entry as a fermion so the main edge receives one particle-flow arrow that follows `edge.orientation`. This is separate from optional momentum arrows.

15.2.23 wave `dictionary`

Photon-style wave pattern.

15.2.24 coil `dictionary`

Gluon-style coil pattern.

15.2.25 zigzag `dictionary`

Weak-boson-style zigzag pattern.

15.2.26 default-edge `dictionary`

Fallback style entry used when an edge has no known particle type.

15.2.27 default-map `dictionary`

Small built-in particle map for standalone physics diagrams. GammaLoop generated `edge-style.typ` files pass their model-specific map to `style`.

15.2.28 momentum-arrow-defaults `dictionary`

Default style for source-to-sink momentum arrow layers.

16 subgraph module

Use `subgraph` to construct and inspect opaque zero-copy selections of half-edges. Pass those selections to graph queries, drawing, and layout functions that accept a `subgraph` argument.

16.1 Concepts

16.1.1 Subgraphs

Subgraph objects are opaque zero-copy values.

- `subgraph.label(g, label)` constructs a subgraph from a base62 label.
- `subgraph.bits(g, bits)` constructs a subgraph from a boolean hedge array.
- `subgraph.compass(g, compass)` selects half edges with a DOT compass point.
- `subgraph.to-label(sg)` returns the base62 label.
- `subgraph.hedges(sg)` returns included hedge indices.
- `subgraph.contains(sg, hedge)` tests hedge membership.

16.2 subgraph

- label()
- bits()
- compass()
- to-label()
- hedges()
- contains()
- complement()
- contains-edge()
- node-depths()

16.2.1 label

Construct a subgraph object from a base62 label.

```
#let g = graph.build({
  graph.node(<a>)
  graph.node(<b>)
  graph.edge(graph.source(<a>, compass: "e"), graph.sink(<b>))
})
#let east = subgraph.compass(g, "e")
#let same = subgraph.label(g, subgraph.to-label(east))
#subgraph.hedges(same).len()
```

16.2.1.1 Parameters

```
label(
  graph: dictionary,
  label: string
) -> bytes
```

16.2.1.1.1 graph `dictionary`

Graph object whose half-edge set the label refers to.

16.2.1.1.2 label `string`

Base62 subgraph label returned by `to-label` or produced by Linnest.

16.2.2 bits

Construct a subgraph object from a boolean hedge array.

```
#let g = graph.build({
  graph.node(<a>)
  graph.node(<b>)
  graph.edge(graph.source(<a>), graph.sink(<b>))
})
#let first = subgraph.bits(g, (true, false))
#subgraph.contains(first, 0)
```

16.2.2.1 Parameters

```
bits(
  graph: dictionary,
  bits: array
) -> bytes
```

16.2.2.1.1 graph `dictionary`

Graph object whose half-edge order defines the bit array.

16.2.2.1.2 bits `array`

Boolean array selecting half edges by graph half-edge index.

16.2.3 compass

Construct a subgraph object from a DOT compass point such as `"n"` or `"s"`.

```
#let g = graph.build({
  graph.node(<a>)
  graph.node(<b>)
  graph.edge(graph.source(<a>, compass: "e"), graph.sink(<b>))
})
#subgraph.hedges(subgraph.compass(g, "e")).len()
```

16.2.3.1 Parameters

```
compass(
  graph: dictionary,
  compass: string
) -> bytes
```

16.2.3.1.1 graph `dictionary`

Graph object to filter by half-edge compass statement.

16.2.3.1.2 compass `string`

DOT compass point such as `"n"`, `"s"`, `"e"`, or `"w"`.

16.2.4 to-label

Convert a subgraph object to its base62 label.

```
#let g = graph.build({
  graph.node(<a>)
  graph.node(<b>)
  graph.edge(graph.source(<a>), graph.sink(<b>))
})
#subgraph.to-label(subgraph.bits(g, (true, false)))
```

16.2.4.1 Parameters

```
to-label(subgraph: bytes) -> string
```

16.2.4.1.1 subgraph `bytes`

Subgraph object returned by this module or by `graph.cycles`/`graph.forests`.

16.2.5 hedges

Return the hedge indices included in a subgraph object.

```
#let g = graph.build({
  graph.node(<a>)
  graph.node(<b>)
  graph.edge(graph.source(<a>), graph.sink(<b>))
})
#subgraph.hedges(subgraph.bits(g, (true, false)))
```

16.2.5.1 Parameters

```
hedges(subgraph: bytes) -> array
```

16.2.5.1.1 subgraph `bytes`

Subgraph object to inspect.

16.2.6 contains

Test whether a subgraph object includes a hedge.

```
#let g = graph.build({
  graph.node(<a>)
  graph.node(<b>)
  graph.edge(graph.source(<a>), graph.sink(<b>))
})
#subgraph.contains(subgraph.bits(g, (true, false)), 0)
```

16.2.6.1 Parameters

```
contains(
  subgraph: bytes,
  hedge: int
) -> bool
```

16.2.6.1.1 subgraph `bytes`

Subgraph object to inspect.

16.2.6.1.2 hedge `int`

Half-edge index to test.

16.2.7 complement

Return the complement of a subgraph's selected half edges.

16.2.7.1 Parameters

```
complement(
  graph: dictionary,
  selected: bytes
) -> bytes
```

16.2.7.1.1 graph `dictionary`

Graph object whose half-edge order defines the result.

16.2.7.1.2 selected `bytes`

Subgraph object to invert.

16.2.8 contains-edge

Test whether either half-edge of an edge record is selected.

16.2.8.1 Parameters

```
contains-edge(
  subgraph: bytes,
  edge: dictionary
) -> bool
```

16.2.8.1.1 subgraph `bytes`

Subgraph object to inspect.

16.2.8.1.2 edge `dictionary`

Edge record from `graph.edges(...)` or draw callback data.

16.2.9 node-depths

Compute breadth-first node depths inside a selected edge set.

16.2.9.1 Parameters

```
node-depths(
  graph: dictionary,
  selected: bytes,
  root: int
) -> dictionary
```

16.2.9.1.1 graph `dictionary`

Graph object to inspect.

16.2.9.1.2 selected `bytes`

Subgraph whose selected half edges define the traversal edges.

16.2.9.1.3 root `int`

Root node index.

Default: `0`

17 Version history

> **History coverage**
> Linnet and Clinnet have canonical histories. The independently versioned `linnet-py` distribution does not yet have a separate changelog in this repository.

17.1 Available histories

- Linnet Rust library versions
- Clinnet command-line renderer versions
- `linnet-py`: no standalone history yet

The changelog records user-visible graph, parser, drawing, and algorithm changes. In particular, users moving across releases should check index ordering, DOT serialization, subgraph behavior, and feature-gated integration changes rather than assuming a serialized graph or incidental ordering is stable.

17.2 Choosing the matching documentation

The `latest` channel may describe unreleased behavior. Documentation under `snapshots/<tag>` is fixed to a tagged repository revision. Choose a snapshot whose component information lists your Rust crate version, and check the `linnet-py` version separately when using the Python bindings.

17.3 Upgrade checklist

Review changes to index ordering, DOT serialization, subgraph behavior, and enabled features. Re-validate serialized graphs and any code that depends on traversal or iteration order.

17.4 Reproducibility record

Record the repository revision, `linnet` and `linnet-py` versions, enabled features, and the input graph format. These distinguish a library change from a parser, serializer, or binding change.

18 Linnet versions

This page renders the canonical Linnet changelog source as part of the Linnet website, navigation, and search.

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog, and this project adheres to Semantic Versioning.

19.1 Unreleased

19.2 0.17.0 - 2026-01-28

19.2.1 Other

- force-directed layout
- make the powerset iterator properly size the subset
- upgrade to all spanning *forests* generation
- Refactor PowersetIterator to support generic ID type
- double ended iterator for subsets
- remove unused crates

19.3 0.16.1 - 2026-01-13

19.3.1 Other

- Quote string values in graph and parser outputs
- revert edge, node and hedge index order to be direct (not based on sort)

19.4 0.16.0 - 2026-01-08

19.4.1 Other

- Add methods for identifying bridges and non-bridges in subgraphs
- Make `trace_unfold` implementable outside of linnet

19.5 0.15.1 - 2026-01-05

19.5.1 Fixed

- fixed ordering problem in merge function and make ordering based on sorting not on direct map

19.5.2 Other

- update quote stripping
- Add agent context files and update project tooling
- Add graph algorithms module with topological sort and transitive ops

19.6 0.15.0 - 2025-12-08

19.6.1 Other

- Refactor Figment handling and improve data merging logic
- *(clinnet)* release v0.1.8

19.7 0.14.1 - 2025-12-03

19.7.1 Other

- Add parallel processing and directional constraints
- update test
- only swap one edge when connecting identities

19.8 0.14.0 - 2025-11-28

19.8.1 Other

- change output name and make it short, and add directional groups

19.9 0.13.3 - 2025-11-24

19.9.1 Other

- update symbolica

19.10 0.13.2 - 2025-11-12

19.10.1 Other

- update to symbolica 0.20
- better templates
- clinnet for nested dots to pdf
- Add Figment integration for configuration handling

19.11 0.13.1 - 2025-11-05

19.11.1 Other

- remove prints

19.12 0.12.0 - 2025-08-18

19.12.1 Other

- all tests pass
- clippy fixes
- write global data outside of `graph[]` scope
- Add ID=ID statements to global data when parsing
- newline after nodes
- dot serialize with name
- write graph name
- correctly parse orientations
- make `iter_edges` iterate with `edge_id` order not involution order

19.13 0.11.0 - 2025-07-29

19.13.1 Other

- serialize global data

19.14 0.10.0 - 2025-07-29

19.14.1 Other

- Add check graph function and extracting by nodes
- Add `is_empty` to Swap trait and update implementations

19.15 0.9.1 - 2025-07-28

19.15.1 Other

- impl asref for newtypes

19.16 0.9.0 - 2025-07-27

19.16.1 Other

- Allow specifying a subgraph in the DOT format.
- Rewrite DOT output to include node, hedge and edge indices and fix bidirectional serialization
- DotGraph has concrete data. Parser now supports global edge and vertex data. Can set edge, vertex and hedge index
- add multi-graph parsing and global graph data

19.17 0.8.0 - 2025-07-14

19.17.1 Other

- clippy fix
- Rename HedgeVec to EdgeVec and introduce HedgeVec
- Add H parameter to HedgeGraph in `iter_edges` and orientation
- add H to SignedCycle
- rename `as_ref` to `to_ref`
- Add `hedge_data` to `hedge_graph`
- add perm by key
- update symbolica
- Fix tree traversal and modify ForestNodeStore API to use Option

19.18 0.7.1 - 2025-05-24

19.18.1 Other

- Fix crown computation by inverting subgraph inclusion check and add inclusion for HedgePair

19.19 0.7.0 - 2025-05-23

19.19.1 Other

- unify iterator signature and remove redundant and misleading functions
- Add zenodo doi

19.20 0.6.3 - 2025-05-23

19.20.1 Other

- improve README

19.21 0.6.2 - 2025-05-22

19.21.1 Other

- remove useless check and set
- Correctly forget identification history and adds subgraph contraction

19.22 0.6.1 - 2025-05-22

19.22.1 Other

- Merge pull request #11 from `alphal00p/jules_wip_3253120278989361716`

19.23 0.6.0 - 2025-05-22

19.23.1 Fixed

- fix all tests and update symbolica
- fixes `forget_identification_history`, If you delete a subgraph you need

19.23.2 Other

- Add crown functions and implement spanning tree enumeration algo based on graph contraction
- PartialEq and Eq for hedgevec
- `from_raw` for hedgevec

19.24 0.5.2 - 2025-05-09

19.24.1 Fixed

- fix involution extraction

19.24.2 Other

- Add Permutation iterators, new graph extract test

19.25 0.5.1 - 2025-05-05

19.25.1 Other

- Clean up debug prints and add cycle test

19.26 0.5.0 - 2025-05-05

19.26.1 Fixed

- fixes all around

19.26.2 Other

- boundary->crown
- hairs->boundary

19.27 0.4.0 - 2025-04-30

19.27.1 Added

- feature flag bincode and serde

19.28 0.3.0 - 2025-04-30

19.28.1 Fixed

- fixed extraction for childvec forest, all tests pass and fix warnings

19.28.2 Other

- adds extract to SmartHedgeVec and involution
- maps now take a FnMut
- derive more (Decode and Encode)

19.29 0.2.3 - 2025-04-08

19.29.1 Other

- unpin serde

19.30 0.2.2 - 2025-04-01

19.30.1 Fixed

- fix test snapshots

19.30.2 Other

- raw image

19.31 0.2.1 - 2025-04-01

19.31.1 Other

- Update README.md, logo is a github asset
- put crates badge

19.32 0.2.0 - 2025-04-01

19.32.1 Fixed

- fix cetz orientation

19.32.2 Other

- Create release-plz.yml

20 Clinnet versions

Clinnet is versioned independently from the Linnet library. Its canonical version history remains the Clinnet changelog source. This rendered copy makes that history part of the manual, navigation, search, and product PDF.

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog, and this project adheres to Semantic Versioning.

21.1 Unreleased

21.2 0.1.8 - 2025-12-04

21.2.1 Other

- redo cli to have subcommands

21.3 0.1.7 - 2025-12-04

21.3.1 Other

- properly forward inputs for a single figure

21.4 0.1.6 - 2025-12-03

21.4.1 Other

- Add parallel processing and directional constraints
- add inputs

21.5 0.1.5 - 2025-11-28

21.5.1 Other

- change output name and make it short, and add directional groups

21.6 0.1.4 - 2025-11-26

21.6.1 Other

- Make figure and grid templates optional with defaults

21.7 0.1.3 - 2025-11-26

21.7.1 Other

- don't overwrite existing templates
- strip quotes strings for typst

21.8 0.1.2 - 2025-11-24

21.8.1 Other

- update symbolica

21.9 0.1.1 - 2025-11-12

21.9.1 Other

- remove unnecessary outputs
- better templates

Python API reference

The supported Python packages available in this version are listed below. The HTML manual provides a separate page for every public class and function; this printable edition keeps a compact inventory. Rust APIs use canonical Rustdoc in the HTML manual.

`linnet-py`

`Exports registered by linnet-py.`

`Cycle`

`Cycle represented as a subgraph and optional loop count.`

`Cycle represented as a subgraph and optional loop count.`

```
class Cycle:
```

Requires features: `extension-module, abi3-py310`

`filter`

Kind: `Getter`

```
def filter(self) -> Subgraph:
```

`Subgraph filter for this cycle.`

`loop_count`

Kind: `Getter`

```
def loop_count(self) -> Optional[int]:
```

`Optional loop count for this cycle.`

`__repr__`

Kind: `Method`

```
def __repr__(self) -> str:
```

`Debug-style representation.`

Exhaustive reference · Source

`DotEdgeData`

`DOT edge data (attributes + optional edge id).`

`DOT edge data (attributes + optional edge id).`

```
class DotEdgeData:
```

Requires features: `extension-module, abi3-py310`

`statements`

Kind: `Getter`

```
def statements(self) -> EdgeStatements:
```

`Edge attributes as a dict-like proxy.`

`local_statements`

Kind: `Getter`

```
def local_statements(self) -> EdgeStatements:
```

`Local edge attributes as a dict-like proxy.`

edge_id

Kind: Getter

```
def edge_id(self) -> Optional[EdgeIndex]:
```

Optional edge id.

__new__

Kind: AssociatedFunction

```
def __new__(cls, statements: Optional[Mapping[str, str]] = None, local_statements:
Optional[Mapping[str, str]] = None, edge_id: Optional[int] = None) -> DotEdgeData:
```

Create edge data from fields.

statements

Kind: Parameter

```
Optional[Mapping[str, str]]
```

Default: None

local_statements

Kind: Parameter

```
Optional[Mapping[str, str]]
```

Default: None

edge_id

Kind: Parameter

```
Optional[int]
```

Default: None

add_statement

Kind: Method

```
def add_statement(self, key: str, value: str) -> None:
```

Insert or update an attribute statement.

key

Kind: Parameter

```
str
```

value

Kind: Parameter

```
str
```

__repr__

Kind: Method

```
def __repr__(self) -> str:
```

Debug-style representation.

Exhaustive reference · Source

DotGraph

DOT-backed hedge graph.

DOT-backed hedge graph.

```
class DotGraph:
```

Requires features: `extension-module, abi3-py310`

global_data

Kind: Getter

```
def global_data(self) -> GlobalData:
```

Global graph attributes.

global_data

Kind: Setter

```
def global_data(self, value: GlobalData) -> None:
```

value

Kind: Parameter

GlobalData

from_string

Kind: AssociatedFunction

```
def from_string(cls, s: str) -> DotGraph:
```

Parse a DOT string into a graph.

s

Kind: Parameter

str

from_string_set

Kind: AssociatedFunction

```
def from_string_set(cls, s: str) -> list[DotGraph]:
```

Parse a DOT string into multiple graphs.

s

Kind: Parameter

str

from_file

Kind: AssociatedFunction

```
def from_file(cls, path: str) -> DotGraph:
```

Parse a DOT file into a graph.

path

Kind: Parameter

str

debug_dot

Kind: Method

```
def debug_dot(self) -> str:
```

Serialize graph to DOT for debugging.

dot

Kind: Method

```
def dot(self) -> str:
```

Serialize the full graph to DOT.

dot_of

Kind: Method

```
def dot_of(self, subgraph: Subgraph) -> str:
```

Serialize a subgraph to DOT.

subgraph

Kind: Parameter

```
Subgraph
```

n_nodes

Kind: Method

```
def n_nodes(self) -> int:
```

Number of nodes.

n_edges

Kind: Method

```
def n_edges(self) -> int:
```

Number of edges.

n_hedges

Kind: Method

```
def n_hedges(self) -> int:
```

Number of hedges.

n_externals

Kind: Method

```
def n_externals(self) -> int:
```

Number of external hedges.

n_internals

Kind: Method

```
def n_internals(self) -> int:
```

Number of internal hedges.

full_filter

Kind: Method

def full_filter(self) -> Subgraph:

Subgraph including all hedges.

empty_subgraph

Kind: Method

def empty_subgraph(self) -> Subgraph:

Empty subgraph of this graph.

iter_edges

Kind: Method

def iter_edges(self) -> list[tuple[HedgePair, EdgeIndex, EdgeData]]:

Iterate edges in the full graph.

iter_edges_of

Kind: Method

def iter_edges_of(self, subgraph: Subgraph) -> list[tuple[HedgePair, EdgeIndex, EdgeData]]:

Iterate edges within a subgraph.

subgraph

Kind: Parameter

Subgraph

iter_nodes

Kind: Method

def iter_nodes(self) -> list[tuple[NodeIndex, list[Hedge], DotVertexData]]:

Iterate nodes in the full graph.

iter_nodes_of

Kind: Method

def iter_nodes_of(self, subgraph: Subgraph) -> list[tuple[NodeIndex, list[Hedge], DotVertexData]]:

Iterate nodes within a subgraph.

subgraph

Kind: Parameter

Subgraph

connected_components

Kind: Method

def connected_components(self, subgraph: Subgraph) -> list[Subgraph]:

Connected components of a subgraph.

subgraph

Kind: Parameter

Subgraph

count_connected_components

Kind: Method

```
def count_connected_components(self, subgraph: Subgraph) -> int:
```

Count connected components.

subgraph

Kind: Parameter

Subgraph

is_connected

Kind: Method

```
def is_connected(self, subgraph: Subgraph) -> bool:
```

Whether a subgraph is connected.

subgraph

Kind: Parameter

Subgraph

depth_first_traverse

Kind: Method

```
def depth_first_traverse(self, subgraph: Subgraph, root_node: NodeIndex,
include_hedge: Optional[Hedge]) -> TraversalTree:
```

Depth-first traversal from a root node.

subgraph

Kind: Parameter

Subgraph

root_node

Kind: Parameter

NodeIndex

include_hedge

Kind: Parameter

Optional[Hedge]

breadth_first_traverse

Kind: Method

```
def breadth_first_traverse(self, subgraph: Subgraph, root_node: NodeIndex,
include_hedge: Optional[Hedge]) -> TraversalTree:
```

Breadth-first traversal from a root node.

subgraph

Kind: Parameter

Subgraph

root_node

Kind: Parameter

NodeIndex

include_hedge
Kind: Parameter

Optional[Hedge]

bridges
Kind: Method

def bridges(self) -> Subgraph:

Bridges in the full graph.

bridges_of
Kind: Method

def bridges_of(self, subgraph: Subgraph) -> Subgraph:

Bridges within a subgraph.

subgraph
Kind: Parameter

Subgraph

cycle_basis
Kind: Method

def cycle_basis(self) -> tuple[list[Cycle], Subgraph]:

Cycle basis of the full graph.

cycle_basis_of
Kind: Method

def cycle_basis_of(self, subgraph: Subgraph) -> tuple[list[Cycle], Subgraph]:

Cycle basis of a subgraph.

subgraph
Kind: Parameter

Subgraph

all_spanning_forests
Kind: Method

def all_spanning_forests(self) -> list[Subgraph]:

All spanning forests of the full graph.

all_spanning_forests_of
Kind: Method

def all_spanning_forests_of(self, subgraph: Subgraph) -> list[Subgraph]:

All spanning forests of a subgraph.

subgraph
Kind: Parameter

Subgraph

combine_to_single_hedgenode

Kind: Method

def combine_to_single_hedgenode(self, nodes: Sequence[NodeIndex]) -> HedgeNode:

Combine nodes into a single hedge node.

nodes

Kind: Parameter

Sequence[NodeIndex]

all_cuts

Kind: Method

def all_cuts(self, source: HedgeNode, target: HedgeNode) -> list[tuple[Subgraph, OrientedCut, Subgraph]]:

All cuts between two hedge nodes.

source

Kind: Parameter

HedgeNode

target

Kind: Parameter

HedgeNode

all_cuts_from_ids

Kind: Method

def all_cuts_from_ids(self, source: Sequence[NodeIndex], target: Sequence[NodeIndex]) -> list[tuple[Subgraph, OrientedCut, Subgraph]]:

All cuts between two sets of node indices.

source

Kind: Parameter

Sequence[NodeIndex]

target

Kind: Parameter

Sequence[NodeIndex]

contract_subgraph

Kind: Method

def contract_subgraph(self, subgraph: Subgraph, node_data_merge: Optional[DotVertexData] = None) -> None:

Contract a subgraph into a single node, deleting its edges.

subgraph

Kind: Parameter

Subgraph

node_data_merge

Kind: Parameter

`Optional[DotVertexData]`

Default: `None`

join

Kind: Method

```
def join(self, other: DotGraph, matching_fn: Any, merge_fn: Any) -> DotGraph:
```

Join two graphs, matching dangling edges via a Python callback.

other

Kind: Parameter

`DotGraph`

matching_fn

Kind: Parameter

`Any`

merge_fn

Kind: Parameter

`Any`

join_mut

Kind: Method

```
def join_mut(self, other: DotGraph, matching_fn: Any, merge_fn: Any) -> None:
```

In-place join, matching dangling edges via a Python callback.

other

Kind: Parameter

`DotGraph`

matching_fn

Kind: Parameter

`Any`

merge_fn

Kind: Parameter

`Any`

extract

Kind: Method

```
def extract(self, subgraph: Subgraph, split_edge_fn: Any, internal_data: Any,
split_node: Any, owned_node: Any) -> DotGraph:
```

Extract a subgraph with Python callbacks to transform edge/node data.

subgraph

Kind: Parameter

Subgraph

split_edge_fn
Kind: Parameter

Any

internal_data
Kind: Parameter

Any

split_node
Kind: Parameter

Any

owned_node
Kind: Parameter

Any

__getitem__
Kind: Method

__getitem__
Kind: Overload

```
def __getitem__(self, key: Hedge) -> DotHedgeData:
```

key
Kind: Parameter

Hedge

__getitem__
Kind: Overload

```
def __getitem__(self, key: NodeIndex) -> DotVertexData:
```

key
Kind: Parameter

NodeIndex

__getitem__
Kind: Overload

```
def __getitem__(self, key: EdgeIndex) -> DotEdgeData:
```

key
Kind: Parameter

EdgeIndex

Exhaustive reference · Source

DotGraphBuilder
Builder for constructing DOT graphs programmatically.

Builder for constructing DOT graphs programmatically.

class DotGraphBuilder:

Requires features: extension-module, abi3-py310

__new__

Kind: AssociatedFunction

def __new__(cls) -> DotGraphBuilder:

Create a new, empty graph builder.

add_node

Kind: Method

def add_node(self, data: Optional[DotVertexData] = None) -> NodeIndex:

Add a node and return its index.

data

Kind: Parameter

Optional[DotVertexData]

Default: None

add_edge

Kind: Method

def add_edge(self, source: NodeIndex, sink: NodeIndex, data: Optional[DotEdgeData] = None, orientation: Optional[Orientation] = None, source_hedge: Optional[DotHedgeData] = None, sink_hedge: Optional[DotHedgeData] = None) -> None:

Add an edge between two nodes.

source

Kind: Parameter

NodeIndex

sink

Kind: Parameter

NodeIndex

data

Kind: Parameter

Optional[DotEdgeData]

Default: None

orientation

Kind: Parameter

Optional[Orientation]

Default: None

source_hedge

Kind: Parameter

Optional[DotHedgeData]

Default: `None`

`sink_hedge`
Kind: `Parameter`

`Optional[DotHedgeData]`

Default: `None`

`add_external_edge`
Kind: `Method`

```
def add_external_edge(self, source: NodeIndex, data: Optional[DotEdgeData] = None,
orientation: Optional[Orientation] = None, flow: Optional[Flow] = None, hedge:
Optional[DotHedgeData] = None) -> None:
```

`Add a dangling (external) edge incident to a node.`

`source`
Kind: `Parameter`

`NodeIndex`

`data`
Kind: `Parameter`

`Optional[DotEdgeData]`

Default: `None`

`orientation`
Kind: `Parameter`

`Optional[Orientation]`

Default: `None`

`flow`
Kind: `Parameter`

`Optional[Flow]`

Default: `None`

`hedge`
Kind: `Parameter`

`Optional[DotHedgeData]`

Default: `None`

`build`
Kind: `Method`

```
def build(self) -> DotGraph:
```

`Build the graph and reset the builder.`

Exhaustive reference · Source

`DotHedgeData`
`DOT hedge (half-edge) data.`

DOT hedge (half-edge) data.

class DotHedgeData:

Requires features: extension-module, abi3-py310

statement
Kind: Getter

def statement(self) -> Optional[str]:

Optional statement string.

id
Kind: Getter

def id(self) -> Optional[Hedge]:

Optional hedge id.

port_label
Kind: Getter

def port_label(self) -> Optional[str]:

Optional port label.

compasspt
Kind: Getter

def compasspt(self) -> Optional[str]:

Optional compass point as a string.

__new__
Kind: AssociatedFunction

def __new__(cls, statement: Optional[str] = None, id: Optional[int] = None, port_label: Optional[str] = None, compasspt: Optional[str] = None) -> DotHedgeData:

Create hedge data from fields.

statement
Kind: Parameter

Optional[str]

Default: None

id
Kind: Parameter

Optional[int]

Default: None

port_label
Kind: Parameter

Optional[str]

Default: None

`compasspt`

Kind: `Parameter`

`Optional[str]`

Default: `None`

`__repr__`

Kind: `Method`

```
def __repr__(self) -> str:
```

`Debug-style representation.`

Exhaustive reference · Source

`DotVertexData`

`DOT vertex data (name, optional index, and attribute statements).`

`DOT vertex data (name, optional index, and attribute statements).`

```
class DotVertexData:
```

Requires features: `extension-module, abi3-py310`

`name`

Kind: `Getter`

```
def name(self) -> Optional[str]:
```

`Optional vertex name.`

`index`

Kind: `Getter`

```
def index(self) -> Optional[NodeIndex]:
```

`Optional node index.`

`statements`

Kind: `Getter`

```
def statements(self) -> NodeStatements:
```

`Attribute statements as a dict-like proxy.`

`__new__`

Kind: `AssociatedFunction`

```
def __new__(cls, name: Optional[str] = None, index: Optional[int] = None, statements:
Optional[Mapping[str, str]] = None) -> DotVertexData:
```

`Create vertex data from fields.`

`name`

Kind: `Parameter`

`Optional[str]`

Default: `None`

`index`

Kind: `Parameter`

`Optional[int]`

Default: `None`

`statements`
Kind: `Parameter`

`Optional[Mapping[str, str]]`

Default: `None`

`add_statement`
Kind: `Method`

`def add_statement(self, key: str, value: str) -> None:`

`Insert or update an attribute statement.`

`key`
Kind: `Parameter`

`str`

`value`
Kind: `Parameter`

`str`

`__repr__`
Kind: `Method`

`def __repr__(self) -> str:`

`Debug-style representation.`

Exhaustive reference · Source

`EdgeData`
`Edge data with orientation.`

`Edge data with orientation.`

`class EdgeData:`

Requires features: `extension-module, abi3-py310`

`orientation`
Kind: `Getter`

`def orientation(self) -> Orientation:`

`Orientation of this edge.`

`data`
Kind: `Getter`

`def data(self) -> DotEdgeData:`

`DOT edge data.`

`__new__`
Kind: `AssociatedFunction`

`def __new__(cls, orientation: Any, data: Any) -> EdgeData:`

`Create edge data from orientation and DotEdgeData.`

orientation

Kind: `Parameter`

`Any`

data

Kind: `Parameter`

`Any`

__repr__

Kind: `Method`

```
def __repr__(self) -> str:
```

Debug-style representation.

Exhaustive reference · Source

EdgeIndex

Edge index identifier.

Edge index identifier.

```
class EdgeIndex:
```

Requires features: `extension-module, abi3-py310`

value

Kind: `Getter`

```
def value(self) -> int:
```

Numeric value of this edge index.

__new__

Kind: `AssociatedFunction`

```
def __new__(cls, value: int) -> EdgeIndex:
```

Create an edge index from a zero-based index.

value

Kind: `Parameter`

`int`

__int__

Kind: `Method`

```
def __int__(self) -> int:
```

Convert to int.

__repr__

Kind: `Method`

```
def __repr__(self) -> str:
```

Debug-style representation.

Exhaustive reference · Source

EdgeStatements

Dict-like proxy for edge attribute statements.

Dict-like proxy for edge attribute statements.

```
class EdgeStatements:
```

Requires features: `extension-module, abi3-py310`

__getitem__

Kind: Method

```
def __getitem__(self, key: str) -> str:
```

key

Kind: Parameter

```
str
```

__setitem__

Kind: Method

```
def __setitem__(self, key: str, value: str) -> None:
```

key

Kind: Parameter

```
str
```

value

Kind: Parameter

```
str
```

__delitem__

Kind: Method

```
def __delitem__(self, key: str) -> None:
```

key

Kind: Parameter

```
str
```

__contains__

Kind: Method

```
def __contains__(self, key: str) -> bool:
```

key

Kind: Parameter

```
str
```

__len__

Kind: Method

```
def __len__(self) -> int:
```

__iter__

Kind: Method

```
def __iter__(self) -> Iterator[str]:
```

get
Kind: Method

```
def get(self, key: str, default: Any = None) -> Any:
```

key
Kind: Parameter

```
str
```

default
Kind: Parameter

```
Any
```

Default: None

keys
Kind: Method

```
def keys(self) -> list[str]:
```

values
Kind: Method

```
def values(self) -> list[str]:
```

items
Kind: Method

```
def items(self) -> list[tuple[str, str]]:
```

clear
Kind: Method

```
def clear(self) -> None:
```

update
Kind: Method

```
def update(self, other: Mapping[str, str]) -> None:
```

other
Kind: Parameter

```
Mapping[str, str]
```

pop
Kind: Method

```
def pop(self, key: str, default: Any = None) -> Any:
```

key
Kind: Parameter

```
str
```

default
Kind: Parameter

Any

Default: None

popitem
Kind: Method

def popitem(self) -> tuple[str, str]:

setdefault
Kind: Method

def setdefault(self, key: str, default: Optional[str] = None) -> str:

key
Kind: Parameter

str

default
Kind: Parameter

Optional[str]

Default: None

__repr__
Kind: Method

def __repr__(self) -> str:

Exhaustive reference · Source

Flow
Flow direction (Source/Sink).

Flow direction (Source/Sink).

class Flow:

Requires features: extension-module, abi3-py310

source
Kind: AssociatedFunction

def source() -> Flow:

Flow::Source.

sink
Kind: AssociatedFunction

def sink() -> Flow:

Flow::Sink.

__repr__
Kind: Method

def __repr__(self) -> str:

Debug-style representation.

Exhaustive reference · Source

GlobalData

Global graph attributes (graph/node/edge statements).

Global graph attributes (graph/node/edge statements).

class GlobalData:

Requires features: extension-module, abi3-py310

name

Kind: Getter

def name(self) -> str:

Graph name.

name

Kind: Setter

def name(self, value: str) -> None:

value

Kind: Parameter

str

statements

Kind: Getter

def statements(self) -> GlobalStatements:

Graph-level statements.

statements

Kind: Setter

def statements(self, value: dict[str, str]) -> None:

value

Kind: Parameter

dict[str, str]

edge_statements

Kind: Getter

def edge_statements(self) -> GlobalStatements:

Default edge statements.

edge_statements

Kind: Setter

def edge_statements(self, value: dict[str, str]) -> None:

value

Kind: Parameter

dict[str, str]

node_statements

Kind: Getter

```
def node_statements(self) -> GlobalStatements:
```

Default node statements.

node_statements

Kind: Setter

```
def node_statements(self, value: dict[str, str]) -> None:
```

value

Kind: Parameter

```
dict[str, str]
```

__new__

Kind: AssociatedFunction

```
def __new__(cls, name: Optional[str] = None, statements: Optional[Mapping[str, str]]
= None, edge_statements: Optional[Mapping[str, str]] = None, node_statements:
Optional[Mapping[str, str]] = None) -> GlobalData:
```

Create global data from fields.

name

Kind: Parameter

```
Optional[str]
```

Default: None

statements

Kind: Parameter

```
Optional[Mapping[str, str]]
```

Default: None

edge_statements

Kind: Parameter

```
Optional[Mapping[str, str]]
```

Default: None

node_statements

Kind: Parameter

```
Optional[Mapping[str, str]]
```

Default: None

add_statement

Kind: Method

```
def add_statement(self, key: str, value: str) -> None:
```

Insert or update a graph-level statement.

key
Kind: `Parameter`

`str`

value
Kind: `Parameter`

`str`

add_edge_statement
Kind: `Method`

```
def add_edge_statement(self, key: str, value: str) -> None:
```

Insert or update a default edge statement.

key
Kind: `Parameter`

`str`

value
Kind: `Parameter`

`str`

add_node_statement
Kind: `Method`

```
def add_node_statement(self, key: str, value: str) -> None:
```

Insert or update a default node statement.

key
Kind: `Parameter`

`str`

value
Kind: `Parameter`

`str`

__repr__
Kind: `Method`

```
def __repr__(self) -> str:
```

Debug-style representation.

Exhaustive reference · Source

GlobalStatements
Dict-like proxy for global statements.

Dict-like proxy for global statements.

```
class GlobalStatements:
```

Requires features: `extension-module, abi3-py310`

__getitem__
Kind: Method

```
def __getitem__(self, key: str) -> str:
```

key
Kind: Parameter

```
str
```

__setitem__
Kind: Method

```
def __setitem__(self, key: str, value: str) -> None:
```

key
Kind: Parameter

```
str
```

value
Kind: Parameter

```
str
```

__delitem__
Kind: Method

```
def __delitem__(self, key: str) -> None:
```

key
Kind: Parameter

```
str
```

__contains__
Kind: Method

```
def __contains__(self, key: str) -> bool:
```

key
Kind: Parameter

```
str
```

__len__
Kind: Method

```
def __len__(self) -> int:
```

__iter__
Kind: Method

```
def __iter__(self) -> Iterator[str]:
```

get
Kind: Method

```
def get(self, key: str, default: Any = None) -> Any:
```

key
Kind: Parameter

str

default
Kind: Parameter

Any

Default: None

keys
Kind: Method

```
def keys(self) -> list[str]:
```

values
Kind: Method

```
def values(self) -> list[str]:
```

items
Kind: Method

```
def items(self) -> list[tuple[str, str]]:
```

clear
Kind: Method

```
def clear(self) -> None:
```

update
Kind: Method

```
def update(self, other: Mapping[str, str]) -> None:
```

other
Kind: Parameter

Mapping[str, str]

pop
Kind: Method

```
def pop(self, key: str, default: Any = None) -> Any:
```

key
Kind: Parameter

str

default
Kind: Parameter

Any

Default: None

popitem
Kind: Method

```
def popitem(self) -> tuple[str, str]:
```

setdefault

Kind: Method

```
def setdefault(self, key: str, default: Optional[str] = None) -> str:
```

key

Kind: Parameter

```
str
```

default

Kind: Parameter

```
Optional[str]
```

Default: `None`

__repr__

Kind: Method

```
def __repr__(self) -> str:
```

Exhaustive reference · Source

Hedge

```
Half-edge identifier.
```

```
Half-edge identifier.
```

```
class Hedge:
```

Requires features: `extension-module, abi3-py310`

value

Kind: Getter

```
def value(self) -> int:
```

```
Numeric value of this hedge.
```

__new__

Kind: AssociatedFunction

```
def __new__(cls, value: int) -> Hedge:
```

```
Create a hedge from a zero-based index.
```

value

Kind: Parameter

```
int
```

__int__

Kind: Method

```
def __int__(self) -> int:
```

```
Convert to int.
```

__repr__

Kind: Method

```
def __repr__(self) -> str:
Debug-style representation.
```

Exhaustive reference · Source

HedgeNode

```
Hedge node with internal graph and hairs.
Hedge node with internal graph and hairs.
class HedgeNode:
```

Requires features: `extension-module, abi3-py310`

internal_graph

Kind: `Getter`

```
def internal_graph(self) -> Subgraph:
Internal subgraph.
```

hairs

Kind: `Getter`

```
def hairs(self) -> Subgraph:
Hair subgraph.
```

__new__

Kind: `AssociatedFunction`

```
def __new__(cls, internal_graph: Subgraph, hairs: Subgraph) -> HedgeNode:
Create a hedge node from internal graph and hairs subgraphs.
```

internal_graph

Kind: `Parameter`

`Subgraph`

hairs

Kind: `Parameter`

`Subgraph`

__repr__

Kind: `Method`

```
def __repr__(self) -> str:
Debug-style representation.
```

Exhaustive reference · Source

HedgePair

```
Hedge pairing (paired, unpaired, split).
Hedge pairing (paired, unpaired, split).
class HedgePair:
```

Requires features: `extension-module, abi3-py310`

kind

Kind: Getter

```
def kind(self) -> str:
Kind of hedge pair: \"paired\", \"unpaired\", or \"split\".
```

source

Kind: Getter

```
def source(self) -> Optional[Hedge]:
Source hedge (paired/split).
```

sink

Kind: Getter

```
def sink(self) -> Optional[Hedge]:
Sink hedge (paired/split).
```

hedge

Kind: Getter

```
def hedge(self) -> Optional[Hedge]:
Hedge (unpaired only).
```

flow

Kind: Getter

```
def flow(self) -> Optional[Flow]:
Flow for unpaired hedge.
```

split

Kind: Getter

```
def split(self) -> Optional[Flow]:
Which side is split for split edges.
```

__repr__

Kind: Method

```
def __repr__(self) -> str:
Debug-style representation.
```

Exhaustive reference · Source

NodeIndex

```
Node index identifier.
Node index identifier.
class NodeIndex:
```

Requires features: `extension-module, abi3-py310`

value

Kind: Getter

```
def value(self) -> int:
Numeric value of this node index.
```

__new__

Kind: `AssociatedFunction`

```
def __new__(cls, value: int) -> NodeIndex:
```

Create a node index from a zero-based index.

value

Kind: `Parameter`

```
int
```

__int__

Kind: `Method`

```
def __int__(self) -> int:
```

Convert to int.

__repr__

Kind: `Method`

```
def __repr__(self) -> str:
```

Debug-style representation.

Exhaustive reference · Source

NodeStatements

Dict-like proxy for node attribute statements.

Dict-like proxy for node attribute statements.

```
class NodeStatements:
```

Requires features: `extension-module, abi3-py310`

__getitem__

Kind: `Method`

```
def __getitem__(self, key: str) -> str:
```

key

Kind: `Parameter`

```
str
```

__setitem__

Kind: `Method`

```
def __setitem__(self, key: str, value: str) -> None:
```

key

Kind: `Parameter`

```
str
```

value

Kind: `Parameter`

```
str
```

__delitem__

Kind: Method

```
def __delitem__(self, key: str) -> None:
```

key

Kind: Parameter

```
str
```

__contains__

Kind: Method

```
def __contains__(self, key: str) -> bool:
```

key

Kind: Parameter

```
str
```

__len__

Kind: Method

```
def __len__(self) -> int:
```

__iter__

Kind: Method

```
def __iter__(self) -> Iterator[str]:
```

get

Kind: Method

```
def get(self, key: str, default: Any = None) -> Any:
```

key

Kind: Parameter

```
str
```

default

Kind: Parameter

```
Any
```

Default: None

keys

Kind: Method

```
def keys(self) -> list[str]:
```

values

Kind: Method

```
def values(self) -> list[str]:
```

items

Kind: Method

```
def items(self) -> list[tuple[str, str]]:
```

clear

Kind: Method

```
def clear(self) -> None:
```

update

Kind: Method

```
def update(self, other: Mapping[str, str]) -> None:
```

other

Kind: Parameter

```
Mapping[str, str]
```

pop

Kind: Method

```
def pop(self, key: str, default: Any = None) -> Any:
```

key

Kind: Parameter

```
str
```

default

Kind: Parameter

```
Any
```

Default: `None`

popitem

Kind: Method

```
def popitem(self) -> tuple[str, str]:
```

setdefault

Kind: Method

```
def setdefault(self, key: str, default: Optional[str] = None) -> str:
```

key

Kind: Parameter

```
str
```

default

Kind: Parameter

```
Optional[str]
```

Default: `None`

__repr__

Kind: Method

```
def __repr__(self) -> str:
```

Exhaustive reference · Source

`Orientation`

Edge orientation (Default/Reversed/Undirected).

```
Edge orientation (Default/Reversed/Undirected).
class Orientation:
```

Requires features: `extension-module, abi3-py310`

`default`

Kind: `AssociatedFunction`

```
def default() -> Orientation:
Orientation::Default.
```

`reversed`

Kind: `AssociatedFunction`

```
def reversed() -> Orientation:
Orientation::Reversed.
```

`undirected`

Kind: `AssociatedFunction`

```
def undirected() -> Orientation:
Orientation::Undirected.
```

`__repr__`

Kind: `Method`

```
def __repr__(self) -> str:
Debug-style representation.
```

Exhaustive reference · Source

`OrientedCut`

Oriented cut represented by left/right subgraphs.

```
Oriented cut represented by left/right subgraphs.
class OrientedCut:
```

Requires features: `extension-module, abi3-py310`

`left`

Kind: `Getter`

```
def left(self) -> Subgraph:
Left side of the cut.
```

`right`

Kind: `Getter`

```
def right(self) -> Subgraph:
Right side of the cut.
```

`__repr__`

Kind: `Method`

```
def __repr__(self) -> str:
```

Debug-style representation.

Exhaustive reference · Source

Subgraph

Subgraph represented as a bitset of hedges.

Subgraph represented as a bitset of hedges.

class Subgraph:

Requires features: extension-module, abi3-py310

empty

Kind: AssociatedFunction

def empty(cls, size: int) -> Subgraph:

Create an empty subgraph of the given size (number of hedges).

size

Kind: Parameter

int

full

Kind: AssociatedFunction

def full(cls, size: int) -> Subgraph:

Create a full subgraph including all hedges.

size

Kind: Parameter

int

from_hedges

Kind: AssociatedFunction

def from_hedges(cls, size: int, hedges: Sequence[Hedge]) -> Subgraph:

Create a subgraph from a list of hedges.

size

Kind: Parameter

int

hedges

Kind: Parameter

Sequence[Hedge]

to_hedges

Kind: Method

def to_hedges(self) -> list[Hedge]:

List included hedges.

size

Kind: Method

```
def size(self) -> int:
```

Total number of hedges in the parent graph.

n_included

Kind: Method

```
def n_included(self) -> int:
```

Number of included hedges.

includes

Kind: Method

```
def includes(self, hedge: Any) -> bool:
```

Whether a hedge is included.

hedge

Kind: Parameter

```
Any
```

union

Kind: Method

```
def union(self, other: Subgraph) -> Subgraph:
```

Union with another subgraph.

other

Kind: Parameter

```
Subgraph
```

intersection

Kind: Method

```
def intersection(self, other: Subgraph) -> Subgraph:
```

Intersection with another subgraph.

other

Kind: Parameter

```
Subgraph
```

sym_diff

Kind: Method

```
def sym_diff(self, other: Subgraph) -> Subgraph:
```

Symmetric difference with another subgraph.

other

Kind: Parameter

```
Subgraph
```

subtract

Kind: Method

```
def subtract(self, other: Subgraph) -> Subgraph:
```

Subtract another subgraph.

other
Kind: Parameter

Subgraph

__repr__
Kind: Method

```
def __repr__(self) -> str:
```

Debug-style representation.

Exhaustive reference · Source

TraversalTree
Traversal tree produced by DFS/BFS.

Traversal tree produced by DFS/BFS.

```
class TraversalTree:
```

Requires features: `extension-module, abi3-py310`

tree_subgraph
Kind: Method

```
def tree_subgraph(self) -> Subgraph:
```

Subgraph corresponding to the tree edges.

node_order
Kind: Method

```
def node_order(self) -> list[NodeIndex]:
```

Node order from traversal.

covers
Kind: Method

```
def covers(self, subgraph: Subgraph) -> Subgraph:
```

Covers a subgraph with the traversal tree.

subgraph
Kind: Parameter

Subgraph

iter_hedges
Kind: Method

```
def iter_hedges(self) -> list[tuple[Hedge, str, Optional[Hedge]]]:
```

Iterate hedges as (hedge, kind, root_hedge).

Exhaustive reference · Source

Version information

Save these identifiers with published results and include them in issue reports so that collaborators can reproduce the same software environment.

- **Documentation channel:** `latest`
- **Documentation route:** `/products/linnet/latest/`
- **Source revision:** `e51747446aa724c3ffac5c42c4103d090c09c6b0`

Package versions and enabled features

- `gammaloop/gammalooprs` — `0.3.4` (`no optional features`)
- `gammaloop/gammaloop-api` — `0.1.0` (`features: cli`)
- `gammaloop/gammaloop` — `0.3.4` (`features: python_api, python_abi, python_stubgen`)
- `linnet/linnet` — `0.17.0` (`features: nodestore-vec, serde, bincode, drawing, symbolica`)
- `linnet/linnet-py` — `0.1.0` (`features: extension-module, abi3-py310`)
- `spenso/spenso` — `0.6.0` (`features: shadowing`)
- `spenso/spenso-macros` — `0.3.2` (`features: shadowing`)
- `spenso/spenso-hep-lib` — `0.1.7` (`no optional features`)
- `spenso/spynso3` — `0.1.2` (`features: python_stubgen`)
- `idenso/idenso` — `0.3.0` (`features: bincode, reference-cases`)
- `idenso/idenso` — `0.3.0` (`features: python, python_stubgen`)
- `vakint/vakint` — `0.1.2` (`no optional features`)
- `vakint/vakint` — `0.1.2` (`features: symbolica_community_module, python_stubgen`)