Package 'splitGraph'

November 5, 2026

Title Dataset Dependency Graphs for Leakage-Aware Evaluation

Version 0.5.0

Description Represent biomedical dataset structure as typed dependency graphs so that sample provenance, repeated-measure structure, study design, batch effects, and temporal relationships are explicit and inspectable. Validates dataset structure, detects sample-level overlap, derives deterministic split constraints, and produces a tool-agnostic split specification for leakage-aware evaluation workflows.

License MIT + file LICENSE

URL https://github.com/selcukorkmaz/splitGraph

BugReports https://github.com/selcukorkmaz/splitGraph/issues

Encoding UTF-8

Depends R (>= 4.1.0)

Imports graphics, igraph, stats, utils

Suggests bioLeak, jsonlite, knitr, pkgload, rmarkdown, rsample, SummarizedExperiment, testthat (>= 3.0.0)

Enhances MultiAssayExperiment

VignetteBuilder knitr

Config/testthat/edition 3

NeedsCompilation no

RoxygenNote 7.3.3

Author Selcuk Korkmaz [aut, cre] (ORCID: <https://orcid.org/0000-0003-4632-6850>)

Maintainer Selcuk Korkmaz <selcukorkmaz@gmail.com>

Repository CRAN

Date/Publication 2026-11-05 11:40:02 UTC

Contents

as_split_spec . . . 2
build_dependency_graph . . . 4
compare_split_constraints . . . 6
create_nodes . . . 7
depgraph_validation_report . . . 9
derive_split_constraints . . . 12
export_graph . . . 14
graph_edit . . . 15
graph_from_metadata . . . 17
graph_node_set . . . 19
ingest_metadata . . . 21
migrate_json . . . 22
pairwise_edges . . . 22
plot.dependency_graph . . . 24
query_node_type . . . 25
splitgraph_conditions . . . 27
validate_json . . . 28
write_dependency_graph . . . 29
write_split_spec . . . 31

Index **33**

`as_split_spec` *Translate splitGraph Constraints into Stable Split Specifications*

Description

Translate graph-derived split constraints into a stable, inspectable structure for sample-level grouping, blocking, and ordering, perform preflight structural checks on that translation, and summarize structural leakage risks.

Usage

```
as_split_spec(
  constraint,
  graph = NULL,
 stratify_by = c("outcome", "none", "batch", "study", "site", "region", "platform",
    "assay")
)

validate_split_spec(x)

summarize_leakage_risks(
  graph,
  constraint = NULL,
  split_spec = NULL,
```

```
  validation = NULL
)
```

Arguments

constraint	A split_constraint.
graph	A dependency_graph.
stratify_by	Optional stratification source when graph is supplied: "outcome" (default, from Outcome nodes), "none" (disable stratum annotation), or one of "batch", "study", "site", "region", "platform", "assay".
x	A split_spec.
split_spec	An optional split_spec.
validation	An optional depgraph_validation_report.

Details

The translation layer always produces canonical sample-level columns including sample_id, sample_node_id, group_id, and primary_group. When available, it also carries the blocking columns (batch_group, study_group, site_group, region_group, platform_group, assay_group), the stratum annotation, and the ordering columns (timepoint_id, time_index, order_rank). Missing but relevant fields are retained as NA columns rather than omitted.

stratum is filled from the graph when one is supplied: the key of the single Outcome node attached to a sample via sample_has_outcome, or, failing that, the single outcome attached to the sample's subject via subject_has_outcome. It is an *annotation* of the outcome level each sample carries, exposed through stratum_var so a downstream consumer (for example scikit-learn's StratifiedGroupKFold) can stratify; splitGraph itself never balances folds. When no sample has a unique outcome, stratum_var is NULL.

When only a subset of samples has ordering metadata, the translated split spec still exposes that partial ordering through time_var, but ordering_required remains FALSE. Ordering is only marked as required when the constraint implies complete ordering coverage.

When graph is supplied, the blocking and ordering annotation columns are filled from the graph wherever the constraint left them NA. This enrichment is best-effort: a source that cannot be resolved unambiguously (for example a sample linked to two batches on a graph built with validate = FALSE) is left as NA and the reason is recorded in metadata$enrichment_warnings (and appended to metadata$warnings) instead of aborting the translation. The primary group_id always comes from the constraint and is never affected.

The split-spec validator checks:

- missing required columns
- missing or duplicated sample identifiers
- missing grouping assignments
- singleton-only grouping structures
- missing ordering when ordering is required
- invalid or empty block variables

Repeated validation of the same split spec yields deterministic issue IDs and diagnostics, which makes the returned validation object stable across runs.

The produced `split_spec` is tool-agnostic. Downstream consumers are expected to provide their own adapters to convert a `split_spec` into their native split representation, so **splitGraph** has no runtime dependency on any of them.

`summarize_leakage_risks()` reuses `validate_graph()` and `split_constraint` metadata rather than duplicating downstream evaluation logic.

Value

`as_split_spec()` returns a `split_spec`. `validate_split_spec()` returns a `split_spec_validation`. `summarize_leakage_risks()` returns a `leakage_risk_summary`.

What downstream consumers read

The `split_spec` contract is wider than any single consumer uses today. Verified against the released versions on 2026-09-14:

Consumer	Reads
bioLeak `as_leaksplits()`	`sample_data` columns `sample_id`, the `group_var` column, `batch_group`, `study_gro`
Python `splitspec` reader (shipped)	every field and column, including `stratum_var` / `stratum` and the block columns
rsample (adapter-cookbook vignette)	`group_id` for `group_vfold_cv()`, `order_rank` for `rolling_origin()`; block columr

Every row is pinned by a contract test (run when bioLeak is installed) so the seam cannot drift silently.

Examples

```
meta <- data.frame(
  sample_id  = c("S1", "S2", "S3", "S4"),
  subject_id = c("P1", "P1", "P2", "P2")
)
g <- graph_from_metadata(meta)

constraint <- derive_split_constraints(g, mode = "subject")
spec <- as_split_spec(constraint, graph = g)
validate_split_spec(spec)
summarize_leakage_risks(g, constraint = constraint, split_spec = spec)
```

`build_dependency_graph` *Assemble and Validate Dependency Graphs*

Description

Combine canonical node and edge tables into a typed dependency graph and perform structural, semantic, and graph-local leakage-aware validation.

Usage

```
build_dependency_graph(
  nodes,
  edges,
  graph_name = NULL,
  dataset_name = NULL,
  validate = TRUE,
  validation_overrides = list()
)

as_igraph(x)

validate_graph(
  graph,
  error_on_fail = FALSE,
  levels = NULL,
  severities = NULL,
  validation_overrides = NULL
)
```

Arguments

nodes, edges	Lists of graph_node_set and graph_edge_set objects.
graph_name, dataset_name	Optional metadata labels.
validate	If TRUE, run validate_graph() before returning.
validation_overrides	Optional named list of explicit validation exceptions. Currently supported keys: allow_multi_subject_samples If TRUE, the semantic validator does not flag samples linked to multiple subjects, and derive_split_constraints(mode = "subject") silently keeps the first listed subject assignment (recording the ambiguity in metadata$warnings). Defaults to FALSE. When passed to validate_graph(), the override is merged into the graph's existing validation_overrides for the duration of the call only.
x	A dependency_graph.
graph	A dependency_graph.
error_on_fail	If TRUE, stop when validation errors are found across all detected issues from the selected validation levels, even if those errors are hidden from issues by severities.
levels	Optional validation layers to run.
severities	Optional severities to retain in the returned issues table. This filter does not change whether the graph is considered valid.

Value

For build_dependency_graph(), a dependency_graph. For validate_graph(), a depgraph_validation_report. For as_igraph(), the underlying igraph object.

Examples

```
meta <- data.frame(
  sample_id = c("S1", "S2"),
  subject_id = c("P1", "P2")
)

samples <- create_nodes(meta, type = "Sample", id_col = "sample_id")
subjects <- create_nodes(meta, type = "Subject", id_col = "subject_id")
edges <- create_edges(
  meta,
  "sample_id",
  "subject_id",
  "Sample",
  "Subject",
  "sample_belongs_to_subject"
)

g <- build_dependency_graph(list(samples, subjects), list(edges))
validate_graph(g)
```

---

compare_split_constraints

Compare Split Constraint Modes and Diagnose Giant-Component Bridges

---

Description

`compare_split_constraints()` evaluates multiple candidate constraint modes on the same `dependency_graph` and returns a side-by-side table of group counts, largest-group concentration, effective group count (inverse Herfindahl–Hirschman index), k-fold feasibility, severed vs. remaining structural leakage warnings, and (when outcome strata are present) the minimum number of distinct groups per outcome stratum.

Usage

```
compare_split_constraints(
  graph,
  modes = NULL,
  k = 5L,
  samples = NULL,
  composite_via = NULL
)

find_dependency_bridges(
  graph,
  via = c("Subject", "Batch", "Study", "Site", "Region", "Platform", "Assay"),
  samples = NULL,
  top_n = NULL
)
```

Arguments

graph	A dependency_graph.
modes	Character vector of constraint modes to compare. When omitted, all modes whose underlying dependency relation is present in graph are compared (plus "composite" when at least two relations exist).
k	Target number of cross-validation folds used to evaluate k_fold_feasible (default 5L).
samples	Optional sample subset (sample IDs or node IDs).
composite_via	Optional via vector passed to derive_split_constraints() when "composite" is evaluated.
via	Dependency node types or pairwise mode names to include when searching for bridging entities in find_dependency_bridges().
top_n	Optional positive integer limiting the number of bridge rows returned.

Details

find_dependency_bridges() identifies non-sample entity nodes (such as shared batches, sites, or subjects) whose removal breaks connected components in the sample-dependency bipartite graph, ranked by the number of components gained and reduction in largest component size.

Value

compare_split_constraints() returns a split_constraint_comparison object (use as.data.frame() for the summary table). find_dependency_bridges() returns a graph_query_result.

Examples

```
meta <- data.frame(
  sample_id  = paste0("S", 1:8),
  subject_id = c("P1", "P1", "P2", "P2", "P3", "P3", "P4", "P4"),
  batch_id   = c("B1", "B1", "B1", "B2", "B2", "B2", "B3", "B3"),
  study_id   = rep(c("ST1", "ST2"), each = 4),
  outcome_id = rep(c("ctrl", "case"), 4)
)
g <- graph_from_metadata(meta)

compare_split_constraints(g, k = 2)
find_dependency_bridges(g, via = c("Subject", "Batch"))
```

---

create_nodes *Create Canonical Node and Edge Tables*

Description

Build canonical node and edge tables from ordinary metadata frames.

Usage

```
create_nodes(
  data,
  type,
  id_col,
  label_col = NULL,
  attr_cols = NULL,
  prefix = TRUE,
  dedupe = TRUE
)

create_edges(
  data,
  from_col,
  to_col,
  from_type,
  to_type,
  relation,
  attr_cols = NULL,
  allow_missing = FALSE,
  dedupe = TRUE,
  from_prefix = TRUE,
  to_prefix = TRUE
)
```

Arguments

`data`	A `data.frame` containing entity or relationship columns.
`type`, `from_type`, `to_type`	Supported node types such as `"Sample"` or `"Subject"`.
`id_col`	Column containing the source identifier for the node type.
`label_col`	Optional column used for node labels.
`attr_cols`	Optional columns stored in the `attrs` list-column.
`prefix`	If `TRUE`, prepend typed prefixes such as `sample:` to node identifiers.
`dedupe`	If `TRUE`, collapse duplicate identifiers or duplicate edges only when the retained definition is identical.
`from_col`, `to_col`	Source and target identifier columns for edge creation.
`relation`	Canonical edge type.
`allow_missing`	If `TRUE`, drop rows with missing edge endpoints instead of erroring.
`from_prefix`, `to_prefix`	Whether to prepend typed prefixes when constructing the edge endpoint identifiers. Defaults preserve the canonical prefixed-ID format.

Details

The package uses typed node identifiers such as `sample:S1` as the canonical graph representation. If you create node sets with `prefix = FALSE`, the corresponding edge endpoints must use matching prefix settings via `from_prefix` and `to_prefix`.

When `dedupe = TRUE`, exact duplicate node or edge definitions are collapsed, but conflicting definitions for the same canonical node identifier or edge relation are rejected with an error.

Value

For `create_nodes()`, a `graph_node_set`. For `create_edges()`, a `graph_edge_set`.

Examples

```
meta <- data.frame(
  sample_id = c("S1", "S2"),
  subject_id = c("P1", "P2")
)

samples <- create_nodes(meta, type = "Sample", id_col = "sample_id")
edges <- create_edges(
  meta,
  from_col = "sample_id",
  to_col = "subject_id",
  from_type = "Sample",
  to_type = "Subject",
  relation = "sample_belongs_to_subject"
)
```

---

depgraph_validation_report

Validation Report Object for splitGraph Graphs

Description

`depgraph_validation_report` is the structured return type produced by `validate_graph()` and `validate_depgraph()`.

Usage

```
depgraph_validation_report(
  graph_name = NULL,
  issues = NULL,
  metrics = list(),
  metadata = list(),
  valid = NULL,
  errors = NULL,
  warnings = NULL,
  advisories = NULL
```

```
)

split_spec(
  sample_data = NULL,
  group_var = "group_id",
  block_vars = character(),
  time_var = NULL,
  stratum_var = NULL,
  ordering_required = FALSE,
  constraint_mode = NULL,
  constraint_strategy = NULL,
  recommended_resampling = NULL,
  metadata = list()
)

split_spec_validation(issues = NULL, metadata = list())

leakage_risk_summary(
  overview = character(),
  diagnostics = NULL,
  validation_summary = list(),
  constraint_summary = list(),
  split_spec_summary = list(),
  metadata = list()
)
```

Arguments

`graph_name`	Graph label stored on the report.
`issues`	Canonical issue table. When `NULL`, an empty skeleton is constructed.
`metrics`	Named list of graph- and issue-level counts.
`metadata`	Named list of report metadata.
`valid`	Optional logical override for the overall validity flag.
`errors`, `warnings`, `advisories`	Optional character vectors of severity-specific messages.
`sample_data`	Sample-level mapping table carried by a `split_spec`.
`group_var`	Name of the grouping column.
`block_vars`	Optional blocking variable names.
`time_var`	Optional ordering column name.
`stratum_var`	Optional name of the column carrying the stratum annotation (the outcome level each sample carries). An annotation only: splitGraph never balances folds.
`ordering_required`	Whether ordering is required for downstream evaluation.
`constraint_mode`, `constraint_strategy`	Constraint-derivation metadata.

`recommended_resampling`
: Optional recommended resampling routine.

`overview`
: Character vector of human-readable overview lines.

`diagnostics`
: Diagnostics data frame for leakage risks.

`validation_summary`, `constraint_summary`, `split_spec_summary`
: Named lists carrying pre-computed summaries.

Details

The report contains:

- `graph_name`: graph label when available
- `valid`: whether any `error`-severity issues were found
- `issues`: canonical issue table
- `summary`: counts by level, severity, and code
- `metadata`: report metadata
- `errors`, `warnings`, `advisories`: backward-compatible message vectors
- `metrics`: graph and issue counts

The canonical issue table includes the columns: `issue_id`, `level`, `severity`, `code`, `message`, `node_ids`, `edge_ids`, and `details`.

Value

An S3 object corresponding to the constructor that was called.

See Also

`validate_graph`

Examples

```
meta <- data.frame(
  sample_id  = c("S1", "S2"),
  subject_id = c("P1", "P2")
)
g <- graph_from_metadata(meta)

report <- validate_graph(g)
report$valid
summary(report)
```

derive_split_constraints *Derive Split Constraints from Dependency Graphs*

Description

Convert dataset dependency structure into deterministic sample-level grouping constraints suitable for leakage-aware evaluation design.

Usage

```
derive_split_constraints(
  graph,
 mode = c("subject", "batch", "study", "time", "site", "region", "platform", "assay",
    "relatedness", "spatial", "composite"),
  samples = NULL,
  strategy = c("strict", "rule_based"),
  via = NULL,
  priority = NULL,
  time_ordering = FALSE,
  min_weight = NULL,
  max_distance = NULL,
  include_warnings = TRUE
)

grouping_vector(x)
```

Arguments

graph	A dependency_graph.
mode	Constraint derivation mode.
samples	Optional sample identifiers or sample node IDs used to restrict the returned sample_map. All requested samples must resolve successfully.
strategy	Composite grouping strategy. Ignored for non-composite modes.
via	Optional dependency sources used for composite grouping. May be given as lower-case modes such as "subject" or node types such as "Subject". Any direct-assignment source ("subject", "batch", "study", "time", "site", "region", "platform", "assay") and either pairwise source ("relatedness", "spatial") can be combined. In the strict strategy a pairwise source contributes its thresholded edges to the same connected-component search as the direct relations; in the rule-based strategy it contributes the component label, and a singleton component counts as "no assignment" so the sample falls through to the next mode. Defaults to c("subject", "batch", "study", "time").
priority	Optional priority order used for strategy = "rule_based".

time_ordering If TRUE, attach temporal ordering metadata (timepoint_id, time_index, order_rank) and set ordering_required = TRUE (when complete) alongside any grouping mode (e.g. longitudinal mode = "subject" or "composite" without collapsing samples at the same visit into a single group).

min_weight Optional numeric minimum weight/kinship threshold applied to subject_related_to edges when mode = "relatedness" or mode = "composite".

max_distance Optional numeric maximum distance threshold applied to sample_adjacent_to edges when mode = "spatial" or mode = "composite".

include_warnings Whether to retain human-readable warnings in the returned metadata.

x A split_constraint.

Details

Constraint derivation rules:

mode = "subject" Groups samples by the target of sample_belongs_to_subject. All samples linked to the same Subject receive the same group_id.

mode = "batch" Groups samples by the target of sample_processed_in_batch. Samples with no batch assignment are retained as singleton unlinked groups and recorded in metadata warnings.

mode = "study" Groups samples by the target of sample_from_study.

mode = "site" Groups samples by the target of sample_collected_at_site. Samples with no site assignment are retained as singleton unlinked groups and recorded in metadata warnings.

mode = "region" Groups samples by the target of sample_located_in_region (e.g. a categorical tissue or anatomical region). Samples with no region assignment are retained as singleton unlinked groups and recorded in metadata warnings.

mode = "platform" Groups samples by the target of sample_run_on_platform (the sequencing / measurement platform or instrument). Samples with no platform assignment are retained as singleton unlinked groups and recorded in metadata warnings.

mode = "assay" Groups samples by the target of sample_measured_by_assay (the assay / modality). Samples with no assay assignment are retained as singleton unlinked groups and recorded in metadata warnings.

mode = "relatedness" Groups samples by transitive closure over thresholded subject_related_to edges (genetic relatedness). Samples that share a subject, or whose subjects are directly or indirectly related above threshold, land in the same connected-component group. Build the edges with relatedness_edges_from_kinship. Samples with no subject are retained as singleton groups (recorded in metadata warnings).

mode = "spatial" Groups samples by transitive closure over thresholded sample_adjacent_to edges (spatial proximity). Build the edges with spatial_edges_from_coords. Isolated samples form singleton groups.

mode = "time" Groups samples by the target of sample_collected_at_timepoint. When Timepoint nodes have time_index metadata, that value is used to derive order_rank. If time_index is unavailable, the function attempts to derive ordering from timepoint_precedes edges over the timepoint subgraph.

`mode = "composite", strategy = "strict"` Projects the selected dependency relations onto a sample graph and assigns one `group_id` per connected component. This is the transitive-closure interpretation of composite dependency grouping.

`mode = "composite", strategy = "rule_based"` Evaluates dependency assignments in deterministic priority order and groups each sample by the highest-priority available dependency source. Lower-priority available dependencies are retained in the explanation field.

The returned `split_constraint$sample_map` always contains `sample_id`, `sample_node_id`, `group_id`, `constraint_type`, `group_label`, and `explanation`. Time-aware constraints also include `time_index`, `timepoint_id`, and `order_rank` when available.

Ambiguous direct assignments are rejected. A sample cannot be assigned to multiple batches, studies, or timepoints when deriving direct split constraints.

Value

`derive_split_constraints()` returns a `split_constraint` whose `sample_map` contains grouping assignments and, for time-aware constraints, ordering metadata. `grouping_vector()` returns a named character vector of `group_id` values keyed by `sample_id`.

Examples

```
meta <- data.frame(
  sample_id  = c("S1", "S2", "S3", "S4"),
  subject_id = c("P1", "P1", "P2", "P2"),
  batch_id   = c("B1", "B2", "B1", "B2")
)
g <- graph_from_metadata(meta)

constraint <- derive_split_constraints(g, mode = "subject")
grouping_vector(constraint)
```

---

export_graph *Export a Dependency Graph for Other Tools*

Description

Write a `dependency_graph` as GraphML or GML (readable by Cytoscape, Gephi, networkx, and `igraph::read_graph()`), or as flat CSV tables of nodes or edges. This complements the lossless JSON format of `write_dependency_graph`: the JSON round-trips exactly and is the interchange contract; these exports are for visual inspection and analysis in graph tools and lose nothing but the nesting of attributes.

Usage

```
export_graph(
  graph,
  file,
  format = c("graphml", "gml", "nodes_csv", "edges_csv")
)
```

Arguments

graph	A dependency_graph.
file	Path to write. For the CSV formats this is the single table requested (nodes_csv writes the node table, edges_csv the edge table).
format	One of "graphml", "gml", "nodes_csv", "edges_csv".

Details

Node attributes (the attrs list-column) are flattened into scalar columns named attr_<name>; multi-valued attributes are collapsed with ";". Canonical columns (node_id, node_type, node_key, label; edge_id, edge_type) are written as-is. In GraphML/GML the vertex name is the node_id.

Value

The normalised output path, invisibly.

Examples

```
meta <- data.frame(sample_id = c("S1", "S2"), subject_id = c("P1", "P2"))
g <- graph_from_metadata(meta)
tmp <- tempfile(fileext = ".graphml")
export_graph(g, tmp, format = "graphml")
igraph::vcount(igraph::read_graph(tmp, format = "graphml"))
unlink(tmp)
```

---

graph_edit *Edit Dependency Graphs*

Description

Derive a new dependency_graph from existing ones without rebuilding from node and edge sets: restrict a graph to a subset of samples, take the union of several graphs, or append edge sets to a graph. Every function returns a new, independently validated dependency_graph; the inputs are never modified.

Usage

```
subset_graph(
  graph,
  samples,
  graph_name = NULL,
  dataset_name = NULL,
  validate = TRUE
)

combine_graphs(..., graph_name = NULL, dataset_name = NULL, validate = TRUE)
```

```
add_edges(
  graph,
  edges,
  graph_name = NULL,
  dataset_name = NULL,
  validate = TRUE
)
```

Arguments

graph	A dependency_graph.
samples	Sample identifiers or sample node ids to keep. All must resolve; unknown ids raise a splitgraph_reference_error.
graph_name, dataset_name	Optional labels for the result. When NULL, subset_graph() and add_edges() inherit the input's labels, and combine_graphs() uses the first non-NULL label among its inputs.
validate	If TRUE (default), run validate_graph() on the result and fail on error-severity issues, as build_dependency_graph() does.
...	For combine_graphs(), two or more dependency_graphs (or a single list of them).
edges	A graph_edge_set or a list of them.

Details

subset_graph() keeps the requested Sample nodes, every edge rooted at one of them (a sample_adjacent_to edge is kept only when both samples are kept), the non-sample nodes those edges point to, and, transitively, non-sample nodes reachable from kept nodes through hierarchical non-sample edges (assay_uses_platform, featureset_generated_from_*, subject_has_outcome). timepoint_precedes and subject_related_to edges are kept only between retained timepoints and subjects, so when time_index is absent the ordering of a subset may become partial; derive_split_constraints(mode = "time") reports that in its warnings. Restricting the graph this way has the same semantics as the samples argument of derive_split_constraints: structure that only reaches the subset through excluded samples is dropped.

combine_graphs() takes the union of node and edge tables. Identical rows are collapsed; a node id or an (from, to, edge_type) relation defined differently in two graphs is an error of class splitgraph_ambiguity_error. Edge ids are regenerated per edge type ("<edge_type>:<k>"), since ids from different graphs would collide. Metadata validation_overrides and edge_sources are merged with later graphs taking precedence.

add_edges() appends one or more graph_edge_sets (for example the output of relatedness_edges_from_kinship) to a graph. New edges receive ids that continue the existing numbering of their edge type; existing ids are preserved. Endpoints must already exist in the graph.

Value

A dependency_graph.

Examples

```
meta <- data.frame(
  sample_id  = c("S1", "S2", "S3", "S4"),
  subject_id = c("P1", "P1", "P2", "P3"),
  batch_id   = c("B1", "B1", "B2", "B2")
)
g <- graph_from_metadata(meta, graph_name = "full")

g_sub <- subset_graph(g, samples = c("S1", "S2"))
summary(g_sub)$node_types

pairs <- data.frame(id1 = "P1", id2 = "P2", kinship = 0.25)
g_kin <- add_edges(g, relatedness_edges_from_kinship(pairs, threshold = 0.1))
grouping_vector(derive_split_constraints(g_kin, mode = "relatedness"))

meta2 <- data.frame(sample_id = c("S5", "S6"), subject_id = c("P3", "P4"))
g_all <- combine_graphs(g, graph_from_metadata(meta2))
summary(g_all)$n_nodes
```

---

graph_from_metadata *Build a Dependency Graph Directly from a Metadata Table*

Description

One-shot convenience builder that auto-detects canonical columns in a metadata table, creates the corresponding node and edge sets, optionally derives timepoint ordering from `time_index`, and assembles a `dependency_graph`. Columns that are absent or entirely missing are silently skipped.

Usage

```
graph_from_metadata(meta, ...)

## Default S3 method:
graph_from_metadata(meta, ...)

## S3 method for class 'SummarizedExperiment'
graph_from_metadata(meta, ..., sample_id_col = NULL)

## S3 method for class 'MultiAssayExperiment'
graph_from_metadata(
  meta,
  ...,
  sample_id_col = "colname",
  subject_id_col = "primary",
  assay_id_col = "assay"
)

## S3 method for class 'data.frame'
```

```
graph_from_metadata(
  meta,
  columns = NULL,
  dataset_name = NULL,
  graph_name = NULL,
  outcome_scope = c("sample", "subject"),
  time_precedence = TRUE,
  validate = TRUE,
  validation_overrides = list(),
  ...
)
```

Arguments

meta	A data.frame containing one row per sample and optional canonical columns: sample_id (required), subject_id, batch_id, study_id, timepoint_id, time_index, assay_id, featureset_id, site_id, region_id, platform_id, outcome_id, or outcome_value. Identifier columns may be character, factor, or numeric; they are coerced to character by ingest_metadata().
...	Passed on to the data.frame method.
sample_id_col	For the SummarizedExperiment method: the colData column holding sample identifiers. When NULL (default) a sample_id column is used if present, otherwise the assay column names (colnames(se)) become the sample identifiers. For the MultiAssayExperiment method: the sampleMap column holding assay sample identifiers (default "colname").
subject_id_col, assay_id_col	For the MultiAssayExperiment method: sampleMap columns holding subject ("primary") and assay ("assay") identifiers.
columns	Optional named character vector passed to ingest_metadata() to rename user columns to canonical names.
dataset_name, graph_name	Optional metadata labels.
outcome_scope	Either "sample" (default) or "subject". Controls whether outcome edges attach to samples or subjects.
time_precedence	If TRUE and time_index is present, derive timepoint_precedes edges from the ordering of time_index.
validate	Forwarded to build_dependency_graph().
validation_overrides	Forwarded to build_dependency_graph().

Details

graph_from_metadata() is an S3 generic. The data.frame method is the one described above. The SummarizedExperiment method (used when Bioconductor's **SummarizedExperiment** is installed) converts colData(se) to a data frame, adds sample_id from the assay column names when colData has no such column, and dispatches to the data.frame method; columns maps colData

names to the canonical ones exactly as for a data frame. A worked example is in `vignette("faq-design-notes")`; it is kept out of the examples below because attaching Bioconductor packages dominates their run time.

Value

A validated dependency_graph.

Examples

```
meta <- data.frame(
  sample_id  = c("S1", "S2", "S3", "S4"),
  subject_id = c("P1", "P1", "P2", "P2"),
  batch_id   = c("B1", "B2", "B1", "B2"),
  timepoint_id = c("T1", "T2", "T1", "T2"),
  time_index = c(1, 2, 1, 2),
  outcome_id = c("ctrl", "case", "ctrl", "case")
)

g <- graph_from_metadata(meta, graph_name = "demo")
g
```

---

graph_node_set *Construct Core splitGraph S3 Objects*

Description

Low-level constructors for the core S3 classes used throughout **splitGraph**.

Usage

```
graph_node_set(
  data = NULL,
  schema_version = .depgraph_schema_version,
  source = list()
)

graph_edge_set(
  data = NULL,
  schema_version = .depgraph_schema_version,
  source = list()
)

dependency_graph(nodes, edges, graph, metadata = list(), caches = list())

graph_query_result(
  query = "",
  params = list(),
```

```
  nodes = NULL,
  edges = NULL,
  table = NULL,
  metadata = list()
)

split_constraint(
  strategy,
  sample_map,
  recommended_downstream_args = list(),
  metadata = list()
)
```

Arguments

data	A data frame matching the canonical schema for nodes or edges.
schema_version	Schema version string stored on the object.
source	Optional source metadata.
nodes, edges	A graph_node_set and graph_edge_set.
graph	An internal igraph object.
metadata, caches, params, recommended_downstream_args	Named lists with auxiliary metadata.
query	Query label stored on a graph_query_result.
table	Tabular query result payload.
strategy	Split strategy identifier.
sample_map	Sample-level mapping table for constraints.

Value

An S3 object corresponding to the constructor that was called.

Examples

```
meta <- data.frame(
  sample_id = c("S1", "S2"),
  subject_id = c("P1", "P2")
)

samples <- create_nodes(meta, type = "Sample", id_col = "sample_id")
subjects <- create_nodes(meta, type = "Subject", id_col = "subject_id")
edges <- create_edges(
  meta,
  from_col = "sample_id",
  to_col = "subject_id",
  from_type = "Sample",
  to_type = "Subject",
  relation = "sample_belongs_to_subject"
)
```

```
nodes_set <- graph_node_set(rbind(samples$data, subjects$data))
edges_set <- graph_edge_set(edges$data)
nodes_set
edges_set
```

`ingest_metadata` *Standardize Sample Metadata*

Description

Normalize user-provided metadata into the canonical column contract used by **splitGraph**.

Usage

```
ingest_metadata(data, col_map = NULL, dataset_name = NULL, strict = TRUE)
```

Arguments

`data`	A sample-level `data.frame`.
`col_map`	Optional named character vector mapping canonical names to user-provided columns.
`dataset_name`	Optional dataset label stored as an attribute on the returned table.
`strict`	If `TRUE`, error when required columns are missing.

Value

A standardized `data.frame` with canonical identifier columns coerced to character.

Examples

```
meta <- ingest_metadata(
  data.frame(sample_id = c("S1", "S2"), subject_id = c("P1", "P2"))
)
```

migrate_json *Upgrade Serialized splitGraph JSON to the Current Schema Version*

Description

Read a dependency_graph or split_spec JSON file written under an older schema_version and rewrite it at the installed version. The round-trip fills any field introduced since the file was written with its default (NA for missing sample_data columns), stamps the current schema_version, and adds the $schema reference. Files already at the current version are rewritten unchanged.

Usage

```
migrate_dependency_graph_json(path, out = path)

migrate_split_spec_json(path, out = path)
```

Arguments

path	Path to the JSON file to upgrade.
out	Path to write the upgraded file to. Defaults to path (in-place upgrade).

Value

The output path, invisibly.

Examples

```
if (requireNamespace("jsonlite", quietly = TRUE)) {
  meta <- data.frame(sample_id = c("S1", "S2"), subject_id = c("P1", "P2"))
  g <- graph_from_metadata(meta)
  tmp <- tempfile(fileext = ".json")
  write_dependency_graph(g, tmp)
  migrate_dependency_graph_json(tmp)
  unlink(tmp)
}
```

pairwise_edges *Build Pairwise Leakage Edges from Continuous Similarity*

Description

Helpers that turn a continuous, pairwise similarity signal into the thresholded, undirected edges consumed by derive_split_constraints(mode = "relatedness") and derive_split_constraints(mode = "spatial"). Only pairs that pass the threshold become edges; the derivation modes then form groups as connected components over those edges (transitive closure), so a chain of individually below-radius neighbours can still land in one group.

Usage

```
relatedness_edges_from_kinship(
  pairs,
  threshold,
  id1 = "id1",
  id2 = "id2",
  kinship = "kinship"
)

spatial_edges_from_coords(coords, radius, id = "sample_id", coord_cols = NULL)
```

Arguments

pairs	Either a data.frame of subject pairs with two id columns and a metric column (the long format written by KING, GCTA, and most kinship tools), or a square symmetric numeric matrix whose row names are subject ids (e.g. PLINK `--make-rel square` output); a matrix is expanded to its upper-triangle pairs before thresholding.
threshold	Minimum kinship value (inclusive) for a pair to be kept.
id1, id2	Column names in `pairs` holding the two subject ids.
kinship	Column name in `pairs` holding the kinship / relatedness value.
coords	A data.frame with one row per sample: a sample id column plus the numeric coordinate columns.
radius	Maximum distance (inclusive) for two samples to be adjacent.
id	Column name in `coords` holding the sample id.
coord_cols	Character vector of coordinate columns in `coords`. Defaults to every numeric column other than `id`.

Details

`relatedness_edges_from_kinship()` keeps subject pairs whose kinship (or relatedness) coefficient is *at least* `threshold` and emits `subject_related_to` edges (`Subject` -> `Subject`).

`spatial_edges_from_coords()` keeps sample pairs whose Euclidean distance over the coordinate columns is *at most* `radius` and emits `sample_adjacent_to` edges (`Sample` -> `Sample`).

Both return a `graph_edge_set` that can be combined with the other node and edge sets in `build_dependency_graph()`. The passing metric value is carried on each edge as an attribute (`kinship` / `distance`).

Value

A `graph_edge_set`.

Examples

```
pairs <- data.frame(
  id1 = c("P1", "P1", "P2"),
  id2 = c("P2", "P3", "P3"),
```

```
  kinship = c(0.25, 0.02, 0.30)
)
relatedness_edges_from_kinship(pairs, threshold = 0.1)

coords <- data.frame(
  sample_id = c("S1", "S2", "S3"),
  x = c(0, 1, 9),
  y = c(0, 1, 9)
)
spatial_edges_from_coords(coords, radius = 2)
```

`plot.dependency_graph` *Plot a Dependency Graph*

Description

Draw a dependency_graph with node colours by type and, by default, a layered layout that places samples on the bottom row and their dependency targets above them.

Usage

```
## S3 method for class 'dependency_graph'
plot(
  x,
  layout = c("typed", "sugiyama", "auto"),
  focus = c("full", "sample_projection", "ego"),
  node = NULL,
  via = NULL,
  order = 1L,
  node_colors = NULL,
  show_labels = TRUE,
  legend = TRUE,
  legend_position = "topleft",
  ...
)
```

Arguments

x	A dependency_graph.
layout	"typed" (one row per node-type layer), "sugiyama" (igraph's layered layout), "auto" (igraph's default), a layout matrix, or a function of the igraph object returning one. For focus = "sample_projection" the typed layout is replaced by a force-directed one, since every node is a sample.
focus	What to draw. "full" (default) draws the typed graph. "sample_projection" draws only the Sample nodes, joined when they share a dependency target of a type in via: the picture of the grouping that derive_split_constraints(mode = "composite") would produce. "ego" draws the neighbourhood of node up to order steps in either direction.

node	For focus = "ego": the node id (e.g. "sample:S1") at the centre of the neighbourhood.
via	For focus = "sample_projection": dependency node types that link samples. Defaults to Subject, Batch, Study, Timepoint.
order	For focus = "ego": neighbourhood radius in edges.
node_colors	Optional named vector overriding the type palette.
show_labels	Draw node labels.
legend, legend_position	Draw a node-type legend and where.
...	Further arguments passed to igraph's plot method.

Value

x, invisibly. Called for the plot.

Examples

```
meta <- data.frame(
  sample_id = c("S1", "S2", "S3"), subject_id = c("P1", "P1", "P2"),
  batch_id = c("B1", "B2", "B1")
)
g <- graph_from_metadata(meta)
plot(g)
plot(g, focus = "sample_projection", via = "Subject")
plot(g, focus = "ego", node = "subject:P1")
```

---

query_node_type *Query Dependency Graph Structure*

Description

Query graph neighborhoods, typed nodes and edges, path structure, projected sample dependency components, and direct shared dependencies within a dependency_graph.

Usage

```
query_node_type(graph, node_types, ids = NULL)

query_edge_type(graph, edge_types, node_ids = NULL)

query_neighbors(
  graph,
  node_ids,
  edge_types = NULL,
  node_types = NULL,
  direction = c("out", "in", "all")
```

```
)

query_paths(
  graph,
  from,
  to,
  edge_types = NULL,
  node_types = NULL,
  mode = c("out", "in", "all"),
  max_length = NULL
)

query_shortest_paths(
  graph,
  from,
  to,
  edge_types = NULL,
  node_types = NULL,
  mode = c("out", "in", "all")
)

detect_dependency_components(
  graph,
  via = c("Subject", "Batch", "Study", "Timepoint", "Site", "Region", "Platform",
    "Assay", "FeatureSet", "Outcome"),
  edge_types = NULL,
  min_size = 1
)

detect_shared_dependencies(
  graph,
  via = c("Subject", "Batch", "Study", "Timepoint"),
  samples = NULL
)
```

Arguments

graph	A dependency_graph.
node_types	Optional node types used to filter node results or allowed path members.
ids	Optional node identifiers used to further restrict query_node_type().
edge_types	Optional edge types used to filter the traversal graph or edge table.
node_ids, from, to	Node identifiers to use as query seeds or endpoints.
direction, mode	Traversal direction.
max_length	Maximum path length (number of edges) for query_paths(). Defaults to a documented finite cap (8) so that igraph::all_simple_paths() cannot blow

	up on dense graphs. Pass `Inf` to opt out and search exhaustively; pass any non-negative integer for an explicit cap. Negative values and non-numeric inputs are rejected.
`via`	Dependency node types used for sample-level dependency detection.
`min_size`	Minimum component size retained by `detect_dependency_components()`.
`samples`	Optional sample identifiers or sample node IDs used to restrict direct shared-dependency detection. All requested samples must resolve successfully.

Details

When a `samples` subset is supplied, partial matching is not allowed: unknown sample identifiers raise an error rather than being silently dropped.

Value

Each function returns a `graph_query_result`. Use `as.data.frame()` to obtain the tidy result table.

Examples

```
meta <- data.frame(
  sample_id  = c("S1", "S2", "S3"),
  subject_id = c("P1", "P1", "P2"),
  batch_id   = c("B1", "B2", "B1")
)
g <- graph_from_metadata(meta)

query_node_type(g, "Sample")
query_neighbors(g, "sample:S1", direction = "out")
detect_shared_dependencies(g, via = "Subject")
```

`splitgraph_conditions` *Classed Conditions Signalled by splitGraph*

Description

Every error raised by **splitGraph** is a classed condition that inherits from `"splitgraph_error"` (and `"error"`), so callers can handle the package's failures selectively with `tryCatch()` without matching on message text. Each condition also carries a machine-readable `code` field drawn from the same vocabulary as the `code` column of a `depgraph_validation_report` where one applies (for example `"missing_source_node"` or `"sample_multiple_batch_assignments"`), and `NA` otherwise.

Condition classes

`splitgraph_error` Base class of every splitGraph error, including argument checks that do not fall in a category below.

`splitgraph_schema_error` The input violates the typed schema: an unsupported node or edge type, an edge whose endpoints have the wrong node types, a graph with no `Sample` node, or a JSON document that is not the expected splitGraph object.

`splitgraph_reference_error` An identifier does not resolve or is not unique: edge endpoints missing from the node table, duplicated node or edge ids, unknown node or sample ids passed to a query, or a missing edge endpoint value.

`splitgraph_ambiguity_error` The structure admits more than one answer where exactly one is required: conflicting definitions for the same node or edge, or a sample linked to several targets of a single-valued relation when deriving a direct constraint.

`splitgraph_validation_error` `validate_graph(error_on_fail = TRUE)` or `build_dependency_graph(validate = TRUE)` found error-severity issues, or timepoint ordering metadata are inconsistent.

`splitgraph_io_error` A file could not be written or parsed.

Warnings raised by the package carry the class `"splitgraph_warning"`.

Examples

```
meta <- data.frame(sample_id = c("S1", "S2"), subject_id = c("P1", "P2"))
g <- graph_from_metadata(meta)
res <- tryCatch(
  query_neighbors(g, "sample:does-not-exist"),
  splitgraph_reference_error = function(e) e$code
)
res
```

`validate_json` *Validate Serialized splitGraph JSON Against the Shipped Schema*

Description

Check that a JSON file written by `write_dependency_graph()` or `write_split_spec()` conforms to the splitGraph on-disk contract. The formal JSON Schemas (Draft 2020-12) ship in `inst/schema/<schema_version>/` and are referenced from the written JSON via the `$schema` key; these functions apply a dependency-free structural check of the same invariants (required fields, value types, node/edge-type enumerations, and referential integrity of edge endpoints) so a handoff file can be validated without a JSON Schema engine.

Usage

```
validate_graph_json(path)

validate_split_spec_json(path)
```

Arguments

path	Path to a serialized dependency_graph or split_spec JSON file.

Value

A splitgraph_json_report: a list with valid (logical), issues (character vector of failures), the detected object_type, and the schema $id.

Examples

```
if (requireNamespace("jsonlite", quietly = TRUE)) {
  meta <- data.frame(sample_id = c("S1", "S2"), subject_id = c("P1", "P2"))
  g <- graph_from_metadata(meta)
  tmp <- tempfile(fileext = ".json")
  write_dependency_graph(g, tmp)
  validate_graph_json(tmp)
  unlink(tmp)
}
```

---

write_dependency_graph

Serialize a Dependency Graph to JSON

Description

Write a dependency_graph to a JSON file and read it back. The on-disk format is intentionally simple and stable: it captures the canonical node table, the canonical edge table (each with their list-column of attributes), the graph metadata (including validation_overrides), and the data-model schema_version. The internal igraph representation is not stored; it is rebuilt on read via dependency_graph().

Usage

```
write_dependency_graph(graph, path, pretty = TRUE)

read_dependency_graph(path, validate = FALSE)
```

Arguments

graph	A dependency_graph produced by build_dependency_graph() or graph_from_metadata().
path	Path to write to or read from.
pretty	If TRUE (default), the JSON is indented for human inspection. Set FALSE for a compact representation.

validate — If TRUE, check the file against the shipped schema (validate_graph_json() / validate_split_spec_json()) before parsing and run validate_graph() (for graphs) or validate_split_spec() (for specs) on the result, failing with a classed error on any violation or error-severity issue. The default FALSE loads the object as written, so a graph saved with validate = FALSE or predating a validation rule still loads; use validate = TRUE for files from untrusted or older sources.

Details

This makes split_spec/dependency_graph objects portable across R sessions, and across language boundaries (any consumer that can read JSON can interpret the format).

Value

write_dependency_graph() invisibly returns path. read_dependency_graph() returns a dependency_graph whose node and edge tables are checked for internal consistency with the rebuilt igraph; with the default validate = FALSE it is *not* re-run through validate_graph().

JSON format

```
{
  "$schema": "https://.../inst/schema/0.3.0/dependency_graph.schema.json",
  "splitGraph_object": "dependency_graph",
  "schema_version": "0.3.0",
  "metadata": {
    "graph_name": "...",
    "dataset_name": "...",
    "created_at": "2026-04-29T10:11:12.000000+0000",
    "schema_version": "0.3.0",
    "validation_overrides": { ... },
    "edge_sources": {
      "subject_related_to": { "relation": "...", "from_col": "...",
                              "to_col": "...", "threshold": 0.125,
                              "metric": "kinship" }
    }
  },
  "nodes": [
    { "node_id": "sample:S1", "node_type": "Sample",
      "node_key": "S1", "label": "S1", "attrs": { ... } },
    ...
  ],
  "edges": [
    { "edge_id": "sample_belongs_to_subject:1",
      "from": "sample:S1", "to": "subject:P1",
      "edge_type": "sample_belongs_to_subject", "attrs": { ... } },
    ...
  ]
}
```

Reading a file whose schema_version shares the installed major version loads silently (additive-only differences); a differing major version loads with a warning suggesting migrate_dependency_graph_json(). The written JSON also carries a $schema reference to the formal JSON Schema shipped under inst/schema/<schema_version>/; validate a file against it with validate_graph_json() or by passing validate = TRUE when reading.

Examples

```
if (requireNamespace("jsonlite", quietly = TRUE)) {
  meta <- data.frame(
    sample_id  = c("S1", "S2"),
    subject_id = c("P1", "P2")
  )
  g <- graph_from_metadata(meta, graph_name = "demo")

  tmp <- tempfile(fileext = ".json")
  write_dependency_graph(g, tmp)
  g2 <- read_dependency_graph(tmp)
  identical(g$nodes$data$node_id, g2$nodes$data$node_id)
  unlink(tmp)
}
```

---

write_split_spec *Serialize a Split Specification to JSON*

Description

Write a split_spec to a JSON file and read it back. The on-disk format captures the canonical sample-level table (sample_data) plus all spec-level fields needed by a downstream resampling adapter (group_var, block_vars, time_var, ordering_required, constraint_mode, constraint_strategy, recommended_resampling) and the spec metadata.

Usage

```
write_split_spec(spec, path, pretty = TRUE)

read_split_spec(path, validate = FALSE)
```

Arguments

spec	A split_spec produced by as_split_spec().
path	Path to write to or read from.
pretty	If TRUE (default), the JSON is indented.
validate	If TRUE, check the file against the shipped schema before parsing and run validate_split_spec() on the result, failing with a classed error on any violation or error-severity issue. Defaults to FALSE.

Details

NA values in sample_data are written as JSON null and read back as NA.

Value

write_split_spec() invisibly returns path. read_split_spec() returns a split_spec.

JSON format

```
{
  "$schema": "https://.../inst/schema/0.3.0/split_spec.schema.json",
  "splitGraph_object": "split_spec",
  "schema_version": "0.3.0",
  "group_var": "group_id",
  "block_vars": ["batch_group", "study_group"],
  "time_var": "order_rank",
  "stratum_var": "stratum",
  "ordering_required": false,
  "constraint_mode": "subject",
  "constraint_strategy": "subject",
  "recommended_resampling": "grouped_cv",
  "metadata": { "relations_used": [...], "via": [...], "priority": [...],
                "threshold": null, "warnings": [...], ... },
  "sample_data": [
    { "sample_id": "S1", "group_id": "subject:P1", "stratum": "case", ... },
    ...
  ]
}
```

Vector-valued metadata fields are always written as arrays, even with a single element. stratum_var and the stratum column were added in schema 0.3.0; files written by earlier versions load with stratum filled as NA.

Examples

```
if (requireNamespace("jsonlite", quietly = TRUE)) {
  meta <- data.frame(
    sample_id  = c("S1", "S2"),
    subject_id = c("P1", "P2")
  )
  g <- graph_from_metadata(meta)
  constraint <- derive_split_constraints(g, mode = "subject")
  spec <- as_split_spec(constraint, graph = g)

  tmp <- tempfile(fileext = ".json")
  write_split_spec(spec, tmp)
  spec2 <- read_split_spec(tmp)
  identical(spec$sample_data$group_id, spec2$sample_data$group_id)
  unlink(tmp)
}
```

Index

add_edges *(graph_edit)*, 15
as_igraph *(build_dependency_graph)*, 4
as_split_spec, 2

build_dependency_graph, 4

combine_graphs *(graph_edit)*, 15
compare_split_constraints, 6
create_edges *(create_nodes)*, 7
create_nodes, 7

dependency_graph *(graph_node_set)*, 19
depgraph_validation_report, 9
derive_split_constraints, 12, *16*
detect_dependency_components
(query_node_type), 25
detect_shared_dependencies
(query_node_type), 25

export_graph, 14

find_dependency_bridges
(compare_split_constraints), 6

graph_edge_set *(graph_node_set)*, 19
graph_edit, 15
graph_from_metadata, 17
graph_node_set, 19
graph_query_result *(graph_node_set)*, 19
grouping_vector
(derive_split_constraints), 12

ingest_metadata, 21

leakage_risk_summary
(depgraph_validation_report), 9

migrate_dependency_graph_json
(migrate_json), 22
migrate_json, 22
migrate_split_spec_json *(migrate_json)*,
22

pairwise_edges, 22
plot.dependency_graph, 24

query_edge_type *(query_node_type)*, 25
query_neighbors *(query_node_type)*, 25
query_node_type, 25
query_paths *(query_node_type)*, 25
query_shortest_paths *(query_node_type)*,
25

read_dependency_graph
(write_dependency_graph), 29
read_split_spec *(write_split_spec)*, 31
relatedness_edges_from_kinship, *13*, *16*
relatedness_edges_from_kinship
(pairwise_edges), 22

spatial_edges_from_coords, *13*
spatial_edges_from_coords
(pairwise_edges), 22
split_constraint *(graph_node_set)*, 19
split_spec
(depgraph_validation_report), 9
split_spec_validation
(depgraph_validation_report), 9
splitgraph_conditions, 27
subset_graph *(graph_edit)*, 15
summarize_leakage_risks
(as_split_spec), 2

validate_graph, *11*
validate_graph
(build_dependency_graph), 4
validate_graph_json *(validate_json)*, 28
validate_json, 28
validate_split_spec *(as_split_spec)*, 2
validate_split_spec_json
(validate_json), 28

write_dependency_graph, *14*, 29
write_split_spec, 31