Package 'pagerankr'

September 28, 2026

Type Package

Title Modular Toolkit for PageRank Calculation

Version 0.1.0

Description Provides a set of modular, pipeable functions to calculate PageRank scores from edge lists and redirect reports, common in SEO analysis. Functions handle URL cleaning, redirect resolution, edge deduplication, isolate handling, and PageRank computation using base R for data manipulation and 'igraph' for core PageRank calculation.

License MIT + file LICENSE

Encoding UTF-8

Language en-US

Depends R (>= 4.0.0)

Imports igraph, rurl (>= 3.0.1), utils

Suggests testthat (>= 3.0.0), withr, knitr, rmarkdown, pkgdown, covr, lintr, spelling, goodpractice, shiny, DT, visNetwork, oysteR, rosv

Config/Needs/build local

Config/testthat/edition 3

VignetteBuilder knitr

URL https://pagerankr-63ad30.gitlab.io/, https://gitlab.com/bart-turczynski/pagerankr

BugReports https://gitlab.com/bart-turczynski/pagerankr/-/issues

Config/roxygen2/version 8.0.0

NeedsCompilation no

Author Bart Turczynski [aut, cre] (ORCID: <https://orcid.org/0000-0002-8788-7980>)

Maintainer Bart Turczynski <bartek@turczynski.pl>

Repository CRAN

Date/Publication 2026-09-28 08:50:02 UTC

Contents

`aggregate_edges` *Aggregate Duplicate Edges After Folding*

Description

Collapses duplicate ‘from -> to‘ rows in an edge list using explicit, per-column aggregation semantics. This is the post-fold aggregation step intended to run *after* redirect / canonical folding has coalesced URL variants onto their representatives, and it is a loss-aware alternative to [get_unique_edges()].

Where [get_unique_edges()] dedups a ‘from/to‘ pair by **keeping the first row** (silently discarding everything else on the duplicate rows), ‘aggregate_edges()‘ combines the duplicate rows column-by-column. This matters the moment edges carry quantities: click counts, predicted click propensities, repeated links to the same destination, and conflicting follow / nofollow metadata are all preserved or combined deterministically instead of dropped.

Usage

```
aggregate_edges(
  edge_list_df,
  agg = list(),
  nofollow_policy = c("any", "all", "majority", "error"),
  preserve_cols = character(0),
  self_loops = c("drop", "keep"),
  from_col = "from",
  to_col = "to"
)
```

Arguments

edge_list_df	A data frame representing the edge list, with at least the ‘from_col‘ and ‘to_col‘ columns.
agg	A named list of per-column aggregation overrides. Names are column names; values are either a built-in aggregation string or a function. Columns not listed use the defaults described above. Default ‘list()‘.
nofollow_policy	The default conflict policy applied to logical columns that are not explicitly listed in ‘agg‘. One of ‘"any"‘ (default), ‘"all"‘, ‘"majority"‘, or ‘"error"‘. Named ‘nofollow_policy‘ because the nofollow flag is the canonical boolean attribute, but it governs every un-overridden logical column.
preserve_cols	Character vector of columns to keep un-collapsed as per-group list-columns (e.g. placement / position features). Default ‘character(0)‘.
self_loops	How to handle self-loops (‘a -> a‘). One of ‘"drop"‘ (default) or ‘"keep"‘.
from_col	Name of the source-node column. Default ‘"from"‘.
to_col	Name of the target-node column. Default ‘"to"‘.

Details

Default per-column semantics

For every column other than ‘from_col‘ / ‘to_col‘ (and any column named in ‘preserve_cols‘), an aggregation is chosen automatically unless overridden in ‘agg‘:

- **numeric / integer columns** (additive counts and click propensities) are summed. Repeated link instances to the same destination therefore add their propensities together: multiple slots pointing at one target produce more total propensity, which is the correct behavioral reading. - **logical columns** (boolean attributes such as ‘nofollow‘) are resolved with an explicit *conflict policy* (see ‘nofollow_policy‘). They are never silently first-wins. - **all other columns** (character, factor, ...) fall back to ‘"first"‘, which reproduces the legacy keep-first behavior for non-additive identifier-like columns.

Overriding per column

‘agg‘ is a named list mapping a column name to either:

- one of the built-in strings ‘"sum"‘, ‘"mean"‘, ‘"max"‘, ‘"min"‘, ‘"first"‘, ‘"last"‘, ‘"any"‘, ‘"all"‘, ‘"majority"‘, or ‘"error"‘, or - a function taking the vector of grouped values and returning a length-1 value.

The boolean conflict policies (‘"any"‘, ‘"all"‘, ‘"majority"‘, ‘"error"‘) may be applied to any logical column. ‘"error"‘ raises if a ‘from/to‘ group holds conflicting (mixed ‘TRUE‘/‘FALSE‘) values; the others reduce to "any TRUE", "all TRUE", and the majority value (ties resolve to ‘TRUE‘) respectively.

Preserving placement features

Columns named in ‘preserve_cols‘ are **not** collapsed. Each surviving ‘from/to‘ group keeps the individual per-instance values as a list-column (one list element per group, holding that group’s vector of values). This lets placement / position features survive aggregation so a later reasonable-surfer model can use each individual link instance.

Backward compatibility

With no weight or extra columns (a plain ‘from‘/‘to‘ edge list), the result is identical to [get_unique_edges()]: NA edges dropped, self-loops handled per ‘self_loops‘, one row per unique ‘from/to‘ pair, from/to coerced to character.

Value

A data frame with one row per unique ‘from/to‘ pair (self-loops handled per ‘self_loops‘). ‘from_col‘ / ‘to_col‘ are coerced to character; each remaining column is aggregated per its resolved rule; ‘preserve_cols‘ become list-columns. Row order follows first appearance of each ‘from/to‘ pair in the (NA-filtered, self-loop-handled) input.

See Also

[get_unique_edges()] for the lossy keep-first dedup.

Examples

```
# Click counts to the same destination sum instead of being dropped.
edges <- data.frame(
  from = c("A", "A", "B"),
  to = c("B", "B", "C"),
  clicks = c(3, 5, 2),
  nofollow = c(FALSE, TRUE, FALSE)
)
aggregate_edges(edges)

# Require agreement on nofollow, erroring on a conflict.
try(aggregate_edges(edges, nofollow_policy = "error"))

# Preserve placement features as a list-column for later modeling.
edges_pos <- data.frame(
  from = c("A", "A"),
  to = c("B", "B"),
  position = c(1, 7)
)
aggregate_edges(edges_pos, preserve_cols = "position")
```

---

align_prior_to_vertices

Align a Per-URL Prior to a PageRank Vertex Set (TIPR)

Description

Builds a personalization / teleport vector for `igraph::page_rank(personalized = )` from a per-URL external-authority prior (e.g. Ahrefs referring domains), aligned to the *final* graph vertex set. This is the core of TIPR ("topic/true internal PageRank"), where the random surfer's teleport mass is distributed in proportion to external authority instead of uniformly.

The prior URLs are expected to already share the vertex namespace (i.e. canonicalized with the same `rurl` settings and folded through the same redirect map as the edges). [pagerank()] performs

that canonicalization and redirect-fold before calling this function; call it directly only when your prior URLs already match `vertex_names`.

Usage

```
align_prior_to_vertices(
  vertex_names,
  prior_df,
  prior_url_col = "url",
  prior_weight_col = "weight",
  transform = c("none", "log", "percentile", "minmax", "zipf", "rank_linear"),
  alpha = 0,
  exclude_nodes = character(0),
  verbose = TRUE
)
```

Arguments

`vertex_names`	Character vector of the graph's vertex names, in graph order (typically `igraph::V(graph)$name`).
`prior_df`	A data frame with one row per URL carrying a raw authority weight (e.g. referring-domain counts). Multiple rows for the same URL are summed (raw counts are additive — summing happens before any transform).
`prior_url_col`	Name of the URL column in `prior_df`. Default `"url"`.
`prior_weight_col`	Name of the numeric weight column in `prior_df`. Default `"weight"`. **Contract: this must be an additive raw count.** URLs that coalesce (duplicate rows here, redirect variants folded upstream by [pagerank()]) are combined by *summation*, which is only meaningful for counts — quantities that genuinely add when two URLs merge. This makes the prior **source-agnostic**: referring domains (the default metric), links-to-target, or dofollow-only referring domains from Ahrefs are interchangeable swaps, as are equivalent counts from other sources (SEMrush, GA4 entrances, ...). Do **not** pass a calculated authority *score* (Ahrefs UR / DR, or any 0–100 rating): scores do not sum across redirect variants (the correct fold would be `max`), and a per-URL score such as UR is itself a PageRank-flavored metric, so using it as the teleport prior for PageRank is circular.
`transform`	Character, how to shape the raw authority before it becomes teleport mass. Passed to [transform_weights()]; one of `"none"` (default, faithful linear share), `"log"`, `"percentile"`, `"minmax"`, `"zipf"`, `"rank_linear"`. The transform is applied only to vertices that actually carry authority; vertices with no prior contribute zero to the authority component.
`alpha`	Numeric in `[0, 1]`, the mixture weight between a uniform teleport and the authority-weighted teleport: `p = alpha * uniform + (1 - alpha) * authority_share`. `alpha = 0` (default) is pure authority teleport (pages with no external authority get no teleport mass, though they still receive rank via inlinks); `alpha = 1` reproduces standard uniform PageRank. `alpha` is a smoothing knob, *not* a dead-node mechanism — isolate/self-loop handling owns dead nodes.

`exclude_nodes` Character vector of vertex names that must receive **zero** teleport in both components: the synthetic sinks (e.g. `"__pr_waste_sink__"`) and, when [pagerank()] runs with `prior_exclude_waste = TRUE`, the collect-but-cannot-pass class (noindex / robots-blocked / response-dead). Excluded vertices still receive rank through their inlinks; only their teleport share is removed.

`verbose` Logical, whether to emit coverage diagnostics via `message()` (vertices receiving authority, unmatched prior URLs and their dropped weight, and the realized uniform mass fraction). Default `TRUE`.

Details

Alignment proceeds as: sum raw weights per URL -> match onto `vertex_names` (unmatched vertices get raw 0) -> apply `transform` to the vertices that carry authority -> normalize to an authority share -> mix with a uniform-over-real-vertices vector via `alpha` -> normalize to sum 1. Because `igraph` re-normalizes the personalization vector internally, only the *relative* weights matter; normalization here is for interpretability and to make `alpha` and `exclude_nodes` behave predictably.

Value

A numeric vector the same length as `vertex_names`, in the same order, summing to 1 (suitable for `igraph::page_rank(personalized = )`). Excluded vertices get exactly 0. If the prior matches no vertex and `alpha = 0`, the function falls back to a uniform vector over the non-excluded vertices and warns.

See Also

[pagerank()], [transform_weights()]

Examples

```
v <- c("https://x/a", "https://x/b", "https://x/c", "__pr_waste_sink__")
prior <- data.frame(
  url = c("https://x/a", "https://x/b"),
  weight = c(900, 100)
)
# Pure linear authority share; sink excluded
align_prior_to_vertices(v, prior,
  exclude_nodes = "__pr_waste_sink__",
  verbose = FALSE
)
# Compress the dynamic range
align_prior_to_vertices(v, prior,
  transform = "log",
  exclude_nodes = "__pr_waste_sink__", verbose = FALSE
)
# Authority-tilted uniform (every real page keeps a baseline)
align_prior_to_vertices(v, prior,
  alpha = 0.15,
  exclude_nodes = "__pr_waste_sink__", verbose = FALSE
)
```

`analyze_pagerank_grid` *Analyze PageRank Grid Results*

Description

Computes distribution metrics for each model in a [pagerank_grid()] result, producing a one-row-per-model summary. Useful for quickly comparing how different parameter configurations affect the shape of the PageRank distribution.

Usage

```
analyze_pagerank_grid(
  grid_result,
  model_id_col = "model_id",
  pr_col = "pagerank"
)
```

Arguments

`grid_result`	A data frame returned by [pagerank_grid()], with columns ‘model_id‘, a node column, and a PageRank value column.
`model_id_col`	Name of the model identifier column. Default ‘"model_id"‘.
`pr_col`	Name of the PageRank value column. Default ‘"pagerank"‘.

Value

A data frame with one row per model and the following columns:

model_id Model identifier

num_nodes Number of nodes in the model

pr_sum Sum of PageRank scores (1 for standard graphs, less when evaporation or vanish is active)

pr_max Maximum PageRank score

pr_gini Gini coefficient (see [pr_gini()])

pr_entropy Shannon entropy (see [pr_entropy()])

pr_top10_share Share of total PR held by the top 10 percent of nodes (see [pr_top_k_share()])

Examples

```
edges <- data.frame(
  from = c("A", "B", "C", "A"),
  to = c("B", "C", "A", "C")
)
params <- list(
  low = list(damping = 0.5),
  high = list(damping = 0.95)
)
grid <- pagerank_grid(edges, params, clean_edge_urls = FALSE)
analyze_pagerank_grid(grid)
```

`audit_canonicals` *Audit Declared Canonical Links*

Description

Analyses a ‘rel=canonical‘ data frame and returns a diagnostic report covering chain lengths, loops, conflicting sources (a page declaring multiple distinct canonicals), self-referencing canonicals, and terminal destinations. Mirrors [audit_redirects()] for the canonical signal; useful as a pre-flight check before passing ‘canonicals_df‘ to [pagerank()].

Declared canonicals are an **advisory** signal, distinct from enforced 3xx redirects. To see how the two interact – which one wins on a shared source, and which canonicals are ignored because their source also redirects – use [audit_fold()].

Usage

```
audit_canonicals(
  canonicals_df,
  edge_list_df = NULL,
  canonical_from_col = "from",
  canonical_to_col = "to",
  edge_from_col = "from",
  edge_to_col = "to"
)
```

Arguments

`canonicals_df`	A data frame of declared canonical links, pairing a source URL with the canonical it declares.
`edge_list_df`	Optional data frame of edges. If provided, orphaned canonicals (sources not present in the edge list) are identified.
`canonical_from_col, canonical_to_col`	From/to columns in ‘canonicals_df‘. Default ‘"from"‘ / ‘"to"‘.
`edge_from_col, edge_to_col`	From/to columns in ‘edge_list_df‘. Default ‘"from"‘ / ‘"to"‘.

Value

A list with class ‘"canonical_audit"‘ mirroring the structure of [audit_redirects()]: ‘n_rules‘, ‘n_self_refs‘, ‘self_refs‘, ‘n_conflicts‘, ‘conflicts‘, ‘n_loops‘, ‘loops‘, ‘chains‘, ‘max_chain_length‘, and (when ‘edge_list_df‘ is given) ‘orphaned_redirects‘ (orphaned canonical sources).

See Also

[audit_redirects()], [audit_fold()], [build_fold_map()]

Examples

```
canonicals <- data.frame(
  from = c("http://a?x=1", "http://b", "http://c"),
  to = c("http://a", "http://canon", "http://c")
)
audit_canonicals(canonicals)
```

---

audit_fold *Combined Cross-Signal Fold Audit (Redirects + Canonicals)*

Description

Audits how 3xx **redirects** and declared **rel=canonical** links combine into a single fold map, surfacing exactly where the two signals interact. Wraps [audit_redirects()] and [audit_canonicals()] for the per-signal views and adds the cross-signal tables from [build_fold_map()]: same-source disagreements, canonicals ignored because their source also redirects, and the 'canonical_conflict_policy' outcome.

Disagreements are never silently resolved – they are always reported here, regardless of which policy decides the winner.

Usage

```
audit_fold(
  redirects_df = NULL,
  canonicals_df = NULL,
  edge_list_df = NULL,
  redirect_from_col = "from",
  redirect_to_col = "to",
  canonical_from_col = "from",
  canonical_to_col = "to",
 duplicate_from_policy = c("strict", "first_wins", "last_wins", "most_frequent",
    "prune_source", "resolve_if_consistent"),
  loop_handling = c("error", "prune_loop", "break_arrow"),
  canonical_duplicate_from_policy = c("strict", "first_wins", "last_wins",
    "most_frequent", "prune_source", "resolve_if_consistent"),
  canonical_loop_handling = c("error", "prune_loop", "break_arrow"),
  canonical_conflict_policy = c("redirect_wins", "error", "canonical_wins"),
  edge_from_col = "from",
  edge_to_col = "to"
)
```

Arguments

redirects_df	Optional data frame of 3xx redirect rules, or 'NULL'.
canonicals_df	Optional data frame of declared rel=canonical links, or 'NULL'. Each row pairs a source URL with the canonical it declares.

`edge_list_df` Optional edge list, passed to the per-signal audits for orphan detection.

`redirect_from_col`, `redirect_to_col`
From/to columns in ‘redirects_df‘. Default ‘"from"‘ / ‘"to"‘.

`canonical_from_col`, `canonical_to_col`
From/to columns in ‘canonicals_df‘. Default ‘"from"‘ / ‘"to"‘.

`duplicate_from_policy`
How to handle a redirect source with multiple distinct targets. See [resolve_redirects()]. Default ‘"strict"‘.

`loop_handling` How to handle redirect cycles. See [resolve_redirects()]. Default ‘"error"‘. Also governs cross-signal cycles in the composed graph.

`canonical_duplicate_from_policy`
How to handle a canonical source with multiple distinct declared canonicals. Reuses the ‘duplicate_from_policy‘ enum. Default ‘"strict"‘.

`canonical_loop_handling`
How to handle cycles among declared canonicals. Reuses the ‘loop_handling‘ enum. Default ‘"error"‘.

`canonical_conflict_policy`
How to resolve a redirect-vs-canonical disagreement on the **same source** URL. One of:

‘"redirect_wins"‘ (Default) The 3xx redirect wins; a canonical declared on a URL that itself redirects is ignored and flagged, never transferred onto the redirect target.

‘"error"‘ Error when a redirect and a canonical disagree on the same source (after audit context is computed). Sources where the two signals agree do not error.

‘"canonical_wins"‘ The declared canonical wins for that source; still flagged in the audit. The explicit exception to the default ignored-canonical-on-redirecting-source rule.

`edge_from_col`, `edge_to_col`
From/to columns in ‘edge_list_df‘.

Value

A list with class ‘"fold_audit"‘ containing:

redirects The [audit_redirects()] result (or ‘NULL‘).

canonicals The [audit_canonicals()] result (or ‘NULL‘).

conflicts Data frame of same-source redirect-vs-canonical cases: ‘source‘, ‘redirect_to‘, ‘canonical_to‘, ‘disagrees‘, ‘resolution‘.

ignored_canonicals Data frame of canonicals dropped because their source also redirects (populated under ‘"redirect_wins"‘).

conflict_policy The ‘canonical_conflict_policy‘ in effect.

See Also

[audit_redirects()], [audit_canonicals()], [build_fold_map()]

Examples

```
redirects <- data.frame(from = "http://a", to = "http://b")
canonicals <- data.frame(from = "http://a", to = "http://d")
# a redirects to b but also declares canonical d => disagreement
audit_fold(redirects, canonicals)
```

`audit_redirects` *Audit Redirect Rules*

Description

Analyses a redirect data frame and returns a diagnostic report covering chain lengths, loops, conflicting sources, self-referencing redirects, and terminal destinations. Useful as a pre-flight check before running `resolve_redirects` or `pagerank`.

Usage

```
audit_redirects(
  redirects_df,
  edge_list_df = NULL,
  redirect_from_col = "from",
  redirect_to_col = "to",
  edge_from_col = "from",
  edge_to_col = "to"
)
```

Arguments

`redirects_df`	A data frame containing redirect rules.
`edge_list_df`	Optional data frame of edges. If provided, orphaned redirects (rules whose source URL does not appear in the edge list) are identified.
`redirect_from_col`	Character, name of the source column in `redirects_df`. Default `"from"`.
`redirect_to_col`	Character, name of the target column in `redirects_df`. Default `"to"`.
`edge_from_col`	Character, name of the source column in `edge_list_df`. Default `"from"`.
`edge_to_col`	Character, name of the target column in `edge_list_df`. Default `"to"`.

Value

A list with class `"redirect_audit"` containing:

n_rules Total number of redirect rules (after NA removal).

n_self_refs Number of self-referencing redirects (from == to).

self_refs Data frame of self-referencing redirects.

n_conflicts Number of source URLs with conflicting targets.

conflicts Data frame listing each conflicting source and its distinct targets.

n_loops Number of redirect loops detected.

loops List of character vectors, each describing a cycle path.

chains Data frame with columns `from`, `to_final`, and `chain_length` showing the terminal destination and hop count for every source URL.

max_chain_length Maximum chain length found.

orphaned_redirects Data frame of redirect sources not found in the edge list (only when `edge_list_df` is provided).

Examples

```
redirects <- data.frame(
  from = c("A", "B", "C", "D", "D", "E"),
  to = c("B", "C", "final", "X", "Y", "E")
)
audit <- audit_redirects(redirects)
print(audit)

# With an edge list to detect orphaned redirects
edges <- data.frame(from = "Z", to = "A")
audit2 <- audit_redirects(redirects, edge_list_df = edges)
audit2$orphaned_redirects
```

`auto_grid` *Generate Parameter Grid for pagerank_grid()*

Description

Creates a named list of parameter lists suitable for passing to [pagerank_grid()]. Each combination of the supplied parameter values becomes one entry, with an auto-generated model ID describing the configuration.

This is the "exhaustive search" complement to manually specifying a ‘params_grid‘ – it generates all combinations of the parameter values you provide.

Usage

```
auto_grid(...)
```

Arguments

`...`	Named arguments where each value is a vector of options to sweep. Parameter names must match [pagerank()] arguments.

Value

A named list of named lists, ready to pass as ‘params_grid‘ to [pagerank_grid()]. Names are auto-generated from the parameter values (e.g., ‘"damping=0.85_self_loops=drop"‘).

Examples

```
# Generate all combinations of damping and self-loop handling
grid <- auto_grid(damping = c(0.85, 0.95), self_loops = c("drop", "keep"))
str(grid)
# $`damping=0.85_self_loops=drop`
# $`damping=0.85_self_loops=keep`
# $`damping=0.95_self_loops=drop`
# $`damping=0.95_self_loops=keep`

# Use with pagerank_grid()
edges <- data.frame(
  from = c("A", "B"), to = c("B", "A")
)
results <- pagerank_grid(edges, auto_grid(damping = c(0.5, 0.85, 0.95)),
  clean_edge_urls = FALSE
)
```

---

build_fold_map *Build a Composed Fold Map from Redirects and Canonicals*

Description

Composes a single URL fold map from two distinct web signals – 3xx **redirects** and declared **rel=canonical** links – and reports, per folded URL, which signal caused the fold. This is the source of truth that [pagerank()] uses to fold edge endpoints and TIPR prior URLs, and that the downstream ‘semantic‘ bridge consumes to build its ‘graph_fold‘ table without duplicating the composition logic.

The two signals are kept separate internally for auditability and resolved with their own duplicate/loop policies, then composed with explicit precedence (see Details). Self-referential pairs (self-redirects, self-canonicals) are dropped as no-ops.

Usage

```
build_fold_map(
  redirects_df = NULL,
  canonicals_df = NULL,
  redirect_from_col = "from",
  redirect_to_col = "to",
  canonical_from_col = "from",
  canonical_to_col = "to",
  duplicate_from_policy = c("strict", "first_wins", "last_wins", "most_frequent",
    "prune_source", "resolve_if_consistent"),
  loop_handling = c("error", "prune_loop", "break_arrow"),
  canonical_duplicate_from_policy = c("strict", "first_wins", "last_wins",
    "most_frequent", "prune_source", "resolve_if_consistent"),
  canonical_loop_handling = c("error", "prune_loop", "break_arrow"),
  canonical_conflict_policy = c("redirect_wins", "error", "canonical_wins")
)
```

Arguments

`redirects_df`	Optional data frame of 3xx redirect rules, or ‘NULL‘.
`canonicals_df`	Optional data frame of declared rel=canonical links, or ‘NULL‘. Each row pairs a source URL with the canonical it declares.
`redirect_from_col, redirect_to_col`	From/to columns in ‘redirects_df‘. Default ‘"from"‘ / ‘"to"‘.
`canonical_from_col, canonical_to_col`	From/to columns in ‘canonicals_df‘. Default ‘"from"‘ / ‘"to"‘.
`duplicate_from_policy`	How to handle a redirect source with multiple distinct targets. See [resolve_redirects()]. Default ‘"strict"‘.
`loop_handling`	How to handle redirect cycles. See [resolve_redirects()]. Default ‘"error"‘. Also governs cross-signal cycles in the composed graph.
`canonical_duplicate_from_policy`	How to handle a canonical source with multiple distinct declared canonicals. Reuses the ‘duplicate_from_policy‘ enum. Default ‘"strict"‘.
`canonical_loop_handling`	How to handle cycles among declared canonicals. Reuses the ‘loop_handling‘ enum. Default ‘"error"‘.
`canonical_conflict_policy`	How to resolve a redirect-vs-canonical disagreement on the **same source** URL. One of:

- **‘"redirect_wins"‘** (Default) The 3xx redirect wins; a canonical declared on a URL that itself redirects is ignored and flagged, never transferred onto the redirect target.
- **‘"error"‘** Error when a redirect and a canonical disagree on the same source (after audit context is computed). Sources where the two signals agree do not error.
- **‘"canonical_wins"‘** The declared canonical wins for that source; still flagged in the audit. The explicit exception to the default ignored-canonical-on-redirecting-source rule.

Details

Composition semantics

1. The redirect rules are resolved to terminal destinations using ‘duplicate_from_policy‘ / ‘loop_handling‘; the canonical rules are resolved **independently** using ‘canonical_duplicate_from_policy‘ / ‘canonical_loop_handling‘. The two terminal maps are kept separate. 2. They are then composed into one graph and resolved to terminals, so that: a **canonical target is itself redirect-resolved** before folding (a canonical may point at a URL that 3xx's), and chains spanning both signals collapse to a single representative. 3. For the **same source**, ‘canonical_conflict_policy‘ decides the winner. Under the default ‘"redirect_wins"‘, the canonical declared on a redirecting source is dropped and recorded in the audit.

Inputs are expected to be **already canonicalized** to the node namespace (e.g. via the same ‘rurl‘ profile used for edges). [pagerank()] cleans redirects and canonicals before composing; call this directly only when your URLs already share that namespace.

Value

A data frame with one row per folded source URL (rows where the URL actually changes), with columns:

from The source URL.

to Its final composed representative.

signal Which signal folded this source: ‘"redirect"‘ or ‘"canonical"‘.

The data frame additionally carries the cross-signal conflict tables as attributes ‘"conflicts"‘ and ‘"ignored_canonicals"‘ (see [audit_canonicals()] / ‘audit_fold()‘).

See Also

[resolve_redirects()], [audit_canonicals()], [pagerank()]

Examples

```
redirects <- data.frame(from = "http://a", to = "http://b")
canonicals <- data.frame(from = "http://c", to = "http://a")
# c declares canonical a, a redirects to b => c folds to b via both signals
build_fold_map(redirects, canonicals)
```

`canonical_profile` *pagerankr URL Canonicalization Profile*

Description

The explicit set of ‘rurl‘ canonicalization arguments that determine a pagerankr node identity (scheme + host + path). Every knob that *shapes that key* is pinned here with an explicit value, so node keys never depend on ‘rurl‘’s own defaults – which have changed across ‘rurl‘ versions (e.g. ‘case_handling‘ flipped from ‘"keep"‘ to ‘"lower_host"‘) and previously desynced the pagerankr <-> semantic join. Knobs that only govern components the key drops (port, query, fragment, userinfo) are deliberately left unpinned – see the "Knobs deliberately left unpinned" note in @details.

Usage

```
canonical_profile()
```

Details

The canonical node key is **scheme + host + non-default port + path + contentful query**, with the path normalized under the WHATWG URL standard and percent-encoding preserved byte-for-byte. Fragment and userinfo are dropped by ‘get_clean_url‘ and identify no resource – userinfo under ‘credential_handling = "strip"‘, which is pinned because the alternative (‘"reject"‘) yields ‘NA‘ rather than a key.

The governing principle: parse, do not fold. pagerankr has redirects and canonical tags as first-class inputs, and those are the site’s own statement about which URLs are the same page.

Canonicalization therefore normalizes only what the *standard* says is the same resource, and asserts nothing about site configuration. If ‘www.example.com/a‘ and ‘example.com/a‘ have no redirect and no canonical between them, they are two nodes – and that is a finding, not a defect to be papered over. The knobs that would fold them (‘www_handling‘, ‘trailing_slash_handling‘, ‘index_page_handling‘, ‘protocol_handling‘) are all pinned to their non-folding values for exactly this reason.

Two corollaries that are easy to get backwards: * A **non-default port is a different origin**, so it stays in the key. ‘rurl‘’s default ‘port_handling = "exclude"‘ drops every port and would merge ‘host:8080‘ with ‘host‘. ‘"strip_default"‘ removes only ‘:80‘ on http and ‘:443‘ on https, which the standard makes redundant. * An **IDN host and its punycode form are the same request on the wire**, so no redirect or canonical can ever fold them – the graph has to. ‘host_encoding = "idna"‘ normalizes both to the punycode form.

Why ‘whatwg‘, and why ‘path_encoding = "keep"‘. These two are the only knobs here that do not simply mirror a ‘rurl‘ default, and they are the load-bearing pair. ‘path_encoding‘ is a *presentation* dial – ‘rurl‘’s own documentation says only ‘"keep"‘ preserves a profile’s canonical identity path verbatim, and that ‘"encode"‘/‘"decode"‘ "may re-encode or decode reserved octets (so ‘ pinned ‘"decode"‘ for several releases and thereby merged ‘/a ‘/a/b‘: two different resources, one node. Identity semantics live on ‘url_standard‘ instead, which reaches a profile-internal path-identity axis no presentation dial can touch.

‘"whatwg"‘ rather than ‘"rfc3986"‘ because pagerankr models what a search engine sees. Measured over the node-key fixture, ‘whatwg‘ additionally resolves percent-encoded dot segments (‘ tab/newline from paths instead of failing the parse, percent-encodes literal spaces instead of failing the parse, and keeps ‘http://host‘ and ‘http://host/‘ as one node. ‘rfc3986‘ splits that last pair and returns ‘NA‘ for the whitespace classes, both of which are common in crawl exports. The cost is that ‘whatwg‘ preserves percent spellings, so ‘/a

The remaining knobs equal ‘rurl‘’s current defaults and are pinned only to freeze them. (The anti-drift guarantee ultimately lives in the golden-key fixtures in ‘test-canonicalization.R‘: ‘rurl‘ 3.0.0 re-keyed six of them by reordering decode after dot-segment removal *without touching any argument*, which the surface guard cannot see.)

All twenty of these arguments are accepted by both ‘rurl::get_clean_url()‘ (the cleaning path) and ‘rurl::safe_parse_url()‘ (the domain-filtering path), so one profile drives both and the two paths stay symmetrical.

Knobs deliberately left unpinned. ‘rurl::get_clean_url()‘ has grown options that govern components the node key does not carry, or that select a parsing route rather than a key component: * ‘source‘ (default ‘"all"‘) – the public-suffix source, reachable only through ‘www_handling‘/‘subdomain_levels_to_keep‘, both pinned to values that do not consult it. * Added in ‘rurl‘ 2.7.0: ‘scheme_policy‘ (default ‘"infer"‘), ‘scheme_acceptance‘ (default ‘"web"‘), ‘engine‘ (default ‘NULL‘) and ‘profile‘ (default ‘NULL‘, an unrelated ‘rurl‘ concept that merely shares a name with this function). In particular ‘profile = "seo"‘ is **not** used and must not be: it bundles ‘protocol_handling = "https"‘, ‘www_handling = "strip"‘, ‘trailing_slash_handling = "strip"‘ and ‘index_page_handling = "strip"‘, every one of which is a redirect class pagerankr resolves through its own auditable fold map ([build_fold_map()], applied by [pagerank()]). Folding those at canonicalization time makes the redirect row self-referential, which the fold map drops as a no-op – the hop is erased before anything can audit or report it.

Under the scheme+host+path key these have no visible effect at their defaults, so pinning them would add noise without changing identity. They are intentionally **not** part of this profile;

instead ‘test-canonicalization.R‘ guards them from two sides, so a ‘rurl‘ change is caught on the pagerankr side rather than silently changing node identity: a behavioral guard asserts that a canonical key really does drop the port, query and fragment (catching a default *flip*), and a surface guard reads ‘rurl‘’s own formals and fails on any argument this profile has neither pinned nor listed above (catching an *addition*). A committed node-key probe covering the cross-platform parse-determinism risk surface pins the keys themselves.

The cross-repo contract requires **semantic** to pin the identical profile; change both repos together.

Accepted divergence on un-canonicalizable input. For a value ‘rurl‘ cannot parse (an unsupported scheme like ‘mailto:‘/‘tel:‘, whitespace, a dotless bare token), ‘rurl‘ returns ‘NA‘. pagerankr’s [clean_url_columns()] keeps such a value as its raw self so it survives as an opaque graph node (see that function; PR #50), whereas semantic’s ‘canonical_url()‘ returns ‘None‘ and drops it (FR-05 rurl byte-parity). This is intentional and does **not** break the ‘node_score‘ <-> ‘page‘ join: valid URLs still produce byte-identical keys on both sides (the actual contract), and in the semantic -> pagerankr bridge semantic canonicalizes and drops un-canonicalizable inputs *before* pagerankr sees the edges, so the raw fallback never fires on that path. It only affects pagerankr run standalone on raw crawl data, where such tokens become opaque nodes instead of being dropped.

Value

A named list of ‘rurl‘ canonicalization arguments.

Examples

```
# The pinned profile that determines pagerankr node identity.
profile <- canonical_profile()
str(profile)

# Key knobs that shape the scheme + host + path node key.
profile$case_handling      # "lower_host"
profile$path_normalization # "dot_segments"
profile$path_encoding      # "keep"    (presentation dial, held at identity)
profile$url_standard       # "whatwg"  (where identity semantics live)
```

`clean_url_columns` *Clean URL Columns in a Data Frame*

Description

Applies ‘rurl::get_clean_url‘ to specified columns of a data frame. URLs are cleaned under pagerankr’s explicit canonicalization profile (see Details), with any arguments in ‘...‘ overriding individual knobs.

Usage

```
clean_url_columns(data_frame, columns = c("from", "to"), ...)
```

Arguments

data_frame	A data frame containing URL columns to be cleaned.
columns	A character vector specifying the names of the columns containing URLs. Defaults to ‘c("from", "to")‘.
...	‘rurl::get_clean_url‘ arguments that override the canonicalization profile per key. Recognized knobs are the ones [canonical_profile()] pins: ‘protocol_handling‘, ‘case_handling‘, ‘www_handling‘, ‘trailing_slash_handling‘, ‘index_page_handling‘, ‘path_normalization‘, ‘scheme_relative_handling‘, ‘subdomain_levels_to_keep‘, ‘host_encoding‘, ‘path_encoding‘, ‘url_standard‘, ‘port_handling‘, ‘query_handling‘, ‘params_keep‘, ‘params_drop‘, ‘params_case_sensitive‘, ‘sort_params‘, ‘empty_param_handling‘, ‘decode_plus‘. Note ‘url_standard‘ governs ‘case_handling‘ and ‘path_normalization‘, so ‘rurl‘ rejects an override of either while the profile pins a standard selector.

Details

The canonicalization profile ([canonical_profile()]) pins every ‘rurl‘ knob explicitly so node identities do not depend on ‘rurl‘’s own (version-dependent) defaults, and keeps the cleaning and domain-filtering paths symmetrical. Most knobs equal ‘rurl‘’s current defaults; six override them because they shape node identity – ‘path_normalization‘, ‘path_encoding‘, ‘url_standard‘, ‘port_handling‘, ‘host_encoding‘ and ‘query_handling‘. In particular ‘path_encoding = "keep"‘ holds that presentation dial at its only identity-preserving value, and ‘url_standard = "whatwg"‘ is where the path-identity semantics actually live. See [canonical_profile()] for details.

NA values in the specified columns are preserved in the output. Downstream functions in the pagerankr workflow (such as get_unique_edges and pagerank) will automatically drop any edge where either from or to is NA.

Tokens that ‘rurl‘ cannot parse as a URL (e.g. a dotless bare label such as ‘"A"‘, which newer ‘rurl‘ normalizes to NA) are left as their raw input value rather than becoming NA. This keeps unparseable but non-missing node identities as opaque nodes instead of silently dropping them, so an odd URL in a crawl is scored as its own node rather than vanishing. Only genuinely missing (NA) inputs stay NA. This raw-fallback is a deliberate, accepted divergence from the sibling ‘semantic‘ project (which drops such inputs); see [canonical_profile()] for why it does not desync the cross-repo node join.

Value

A data frame with the specified URL columns cleaned.

Examples

```
df <- data.frame(
  from = c(
    "http://example.com/path",
    "HTTPS://Example.com/PATH#frag", NA,
    "http://example.com/path"
  ),
  to = c(
    "www.another.com?q=1", "another.com/?q=1&b=2",
    "http://foo.bar", NA
```

```
  ),
  other_col = 1:4
)
cleaned_df <- clean_url_columns(df, columns = c("from", "to"))
print(cleaned_df)

# Pass extra arguments to rurl::get_clean_url via ...
cleaned_df_custom <- clean_url_columns(
  df,
  columns = c("from", "to"),
  protocol_handling = "http"
)
print(cleaned_df_custom)
```

---

compare_pagerank *Compare Two PageRank Results*

Description

Performs a full outer join on two PageRank result data frames and computes deltas, percentage changes, and rank changes for each node. Summary statistics are attached as an attribute.

Usage

```
compare_pagerank(
  pr_a,
  pr_b,
  node_col = "node_name",
  pr_col = "pagerank",
  label_a = "a",
  label_b = "b"
)
```

Arguments

pr_a	A data frame of PageRank results (model A / baseline).
pr_b	A data frame of PageRank results (model B / comparison).
node_col	Name of the node column present in both data frames. Default ‘"node_name"‘.
pr_col	Name of the PageRank value column present in both data frames. Default ‘"pagerank"‘.
label_a	Short label for model A (used in output column names). Default ‘"a"‘.
label_b	Short label for model B (used in output column names). Default ‘"b"‘.

Value

A data frame with columns:

node_name Node identifier

pagerank_a, pagerank_b PageRank scores from each model (‘NA‘ when the node is absent from that model)

delta ‘pagerank_b - pagerank_a‘

pct_change Percentage change from a to b (‘NA‘ when a is ‘NA‘ or 0)

rank_a, rank_b Ordinal rank (1 = highest PageRank) within each model (‘NA‘ when the node is absent)

rank_delta ‘rank_a - rank_b‘ (positive = improved in b)

A ‘"summary"‘ attribute (named list) is attached with:

spearman_rho Spearman rank correlation on common nodes

mean_abs_delta Mean of absolute delta on common nodes

nodes_gained Count of nodes in b but not a

nodes_lost Count of nodes in a but not b

Examples

```
pr_a <- data.frame(
  node_name = c("A", "B", "C"),
  pagerank = c(0.5, 0.3, 0.2)
)
pr_b <- data.frame(
  node_name = c("A", "B", "D"),
  pagerank = c(0.4, 0.35, 0.25)
)
result <- compare_pagerank(pr_a, pr_b)
print(result)
attr(result, "summary")
```

`compute_hits` *Compute HITS hub and authority scores using igraph*

Description

Builds a directed graph from a processed edge list and computes Kleinberg’s HITS hub and authority scores using ‘igraph::hits_scores()‘ (the non-deprecated successor of ‘igraph::hub_score()‘ / ‘igraph::authority_score()‘). This is the low-level computational core; the high-level [hits()] wrapper runs the URL-cleaning, redirect/canonical folding, domain filtering, deduplication, and isolate handling identity pipeline first.

Usage

```
compute_hits(
  edge_list_df,
  vertices_df = NULL,
  from_col = "from",
  to_col = "to",
  vertex_col_name = "node_name",
  weight_col = NULL,
  weight_validation = c("error", "warning", "none"),
  scale = TRUE,
  pr_node_col = "node_name",
  hub_col = "hub",
  authority_col = "authority",
  ...
)
```

Arguments

edge_list_df	A data frame representing the processed edge list, with source/target columns (see ‘from_col‘, ‘to_col‘). NAs in those columns are omitted before graph construction.
vertices_df	An optional single-column data frame of node names defining the vertex set (e.g. to retain isolates). If ‘NULL‘ (default), the vertices are inferred from ‘edge_list_df‘. The column name is given by ‘vertex_col_name‘.
from_col, to_col	Names of the source/target columns in ‘edge_list_df‘. Defaults ‘"from"‘ / ‘"to"‘.
vertex_col_name	Name of the node column in ‘vertices_df‘. Default ‘"node_name"‘.
weight_col	Optional name of a numeric edge-weight column. Higher weights give an edge more influence in the hub/authority mutual reinforcement. If ‘NULL‘ (default), the graph is unweighted.
weight_validation	How invalid edge weights are handled when ‘weight_col‘ is supplied: ‘"error"‘ (default), ‘"warning"‘, or ‘"none"‘. See [validate_edge_weights()].
scale	Logical, passed to ‘igraph::hits_scores()‘. When ‘TRUE‘ (default) each score vector is scaled so its maximum entry is ‘1‘, the conventional HITS reporting convention. When ‘FALSE‘ the raw principal eigenvectors (unit Euclidean norm) are returned.
pr_node_col	Name for the node column in the output. Default ‘"node_name"‘ (kept consistent with [compute_pagerank()]).
hub_col, authority_col	Names for the hub and authority score columns in the output. Defaults ‘"hub"‘ / ‘"authority"‘.
...	Additional arguments passed to ‘igraph::hits_scores()‘ (e.g. ‘options‘).

Details

Matrix formulation

Let A be the adjacency matrix of the directed graph ($A_{ij} = 1$ when page i links to page j, or the edge weight when weighted). HITS computes two mutually reinforcing scores as the dominant eigenvectors:

- **authority** is the dominant eigenvector of $A^\top A$: a page is a good authority when it is pointed to by good hubs.
- **hub** is the dominant eigenvector of $AA^\top$: a page is a good hub when it points to good authorities.

‘igraph::hits_scores()‘ solves these eigenproblems directly, so no separate direction flip is needed: authority is the inflow-oriented score and hub is the outflow-oriented score, both returned from a single call.

Value

A data frame with three columns: the node name (named by ‘pr_node_col‘) and the hub and authority scores (named by ‘hub_col‘ / ‘authority_col‘). Returns an empty (zero-row) data frame with those columns when the graph has no vertices.

See Also

[hits()] for the full identity pipeline; [compute_pagerank()] for the PageRank analogue.

Examples

```
edges <- data.frame(
  from = c("A", "A", "B"), to = c("B", "C", "C")
)
compute_hits(edges)

# Retain an isolate via vertices_df (scores 0 for both hub and authority)
verts <- data.frame(node_name = c("A", "B", "C", "D"))
compute_hits(edges, vertices_df = verts)
```

`compute_pagerank` *Compute PageRank using igraph*

Description

Builds a graph from a processed edge list and computes PageRank scores using ‘igraph::page_rank()‘.

Usage

```
compute_pagerank(
  edge_list_df,
  vertices_df = NULL,
  damping = 0.85,
  algo = c("prpack", "arpack"),
  eps = NULL,
  niter = NULL,
  from_col = "from",
  to_col = "to",
  vertex_col_name = "node_name",
  reverse = FALSE,
  weight_col = NULL,
  weight_validation = c("error", "warning", "none"),
  weight_expected_total = NULL,
  weight_tolerance = sqrt(.Machine$double.eps),
  pr_node_col = "node_name",
  pr_value_col = "pagerank",
  prior_df = NULL,
  prior_url_col = "url",
  prior_weight_col = "weight",
  prior_transform = "none",
  prior_alpha = 0,
  prior_exclude_nodes = character(0),
  prior_verbose = TRUE,
  ...
)
```

Arguments

edge_list_df	A data frame representing the processed edge list, typically with columns "from" and "to" (or as specified by ‘from_col‘, ‘to_col‘). It should contain only edges to be included in the graph. NAs in these columns will be omitted before graph construction.
vertices_df	An optional single-column data frame of node names to define the set of vertices for the graph. If ‘NULL‘ (default), all unique non-NA nodes present in ‘edge_list_df‘ (after NA removal from edges) are used. The column name is specified by ‘vertex_col_name‘.
damping	The damping factor for PageRank. A single number in ‘[0, 1]‘, default ‘0.85‘. ‘NA‘, ‘NaN‘ and the infinities are rejected.
algo	Solver back-end passed to ‘igraph::page_rank()‘. Either ‘"prpack"‘ (default; a fast, exact direct solver with no tunable convergence controls) or ‘"arpack"‘ (an iterative eigensolver that honors ‘eps‘ / ‘niter‘ and reports its iteration count). Supplying ‘eps‘ or ‘niter‘ while leaving ‘algo‘ at its default transparently switches to ‘"arpack"‘, since PRPACK ignores those controls. See [pagerank_convergence] for the trade-offs.

`eps` Optional convergence tolerance (L1, the ARPACK ‘options$tol‘). When supplied, the solver switches to ‘"arpack"‘ and iterates until the residual is at or below ‘eps‘. A single finite positive number, or ‘NULL‘ (default) to use the solver’s own default.

`niter` Optional maximum iteration count (the ARPACK ‘options$maxiter‘). When supplied, the solver switches to ‘"arpack"‘. A single whole number from ‘1‘ to ‘.Machine$integer.max‘, or ‘NULL‘ (default) to use the solver’s own default. A fractional value is an error rather than being truncated. As a rule of thumb, power-iteration PageRank needs about ‘log10(eps) / log10(damping)‘ iterations, so raise ‘niter‘ when you raise ‘damping‘ toward 1.

`from_col` Name of the source node column in ‘edge_list_df‘. Default "from".

`to_col` Name of the target node column in ‘edge_list_df‘. Default "to".

`vertex_col_name`
Name of the column in ‘vertices_df‘ containing node names. Default "node_name".

`reverse` Logical. If ‘TRUE‘, edge orientation is flipped before the graph is built, so PageRank is computed on the transposed graph. This is the reverse / inverse PageRank (a.k.a. CheiRank): instead of inflow importance ("who points to me"), it measures outflow centrality ("does this page funnel authority outward"). Vertices, weights, and the teleport prior are unaffected by the flip; only edge direction is reversed. Default ‘FALSE‘. See [pagerank()] for the higher-level wrapper and the caveats on combining ‘reverse = TRUE‘ with direction-sensitive features.

`weight_col` Optional name of a numeric column in ‘edge_list_df‘ containing edge weights. Higher weights make edges more likely to be followed in the random surfer model. If ‘NULL‘ (default), all edges have equal weight (unweighted PageRank).

`weight_validation`
How invalid edge weights are handled when ‘weight_col‘ is supplied: ‘"error"‘ (default), ‘"warning"‘, or ‘"none"‘. Validation covers negative and non-finite values plus sources whose outgoing weights are all zero. See [validate_edge_weights()].

`weight_expected_total`
Optional expected per-source weight total. Leave ‘NULL‘ (default) for ordinary raw edge weights. Set to ‘1‘ when ‘weight_col‘ contains pre-normalized transition probabilities.

`weight_tolerance`
Non-negative tolerance used with ‘weight_expected_total‘.

`pr_node_col` Name for the node column in the output PageRank data frame. Default "node_name".

`pr_value_col` Name for the PageRank value column in the output data frame. Default "pagerank".

`prior_df` Optional per-URL external-authority prior (TIPR). When supplied, a personalization/teleport vector is built via [align_prior_to_vertices()] from the final vertex set and passed to ‘igraph::page_rank(personalized =)‘. The prior URLs must already share the vertex namespace (canonicalized + redirect-folded); [pagerank()] handles that. Default ‘NULL‘ (uniform teleport).

`prior_url_col, prior_weight_col`
Column names in ‘prior_df‘. Defaults ‘"url"‘ / ‘"weight"‘.

prior_transform, prior_alpha, prior_exclude_nodes, prior_verbose
: Passed to [align_prior_to_vertices()] as ‘transform‘, ‘alpha‘, ‘exclude_nodes‘, ‘verbose‘. See that function for semantics. When ‘prior_df‘ is ‘NULL‘, ‘prior_exclude_nodes‘ still applies: the teleport is uniform over the vertices it does not name (empty vector => plain uniform teleport).

...
: Additional arguments passed to ‘igraph::page_rank()‘.

Value

A data frame with two columns: one for node names (named by ‘pr_node_col‘) and one for their PageRank scores (named by ‘pr_value_col‘), which sum to 1 for non-empty graphs. Returns an empty data frame with correct column names if the graph is empty or has no nodes after processing.

For non-empty graphs the result carries a ‘"convergence"‘ attribute (a [pagerank_convergence] object) recording the solver used, iterations (when the solver exposes them), and the post-hoc L1 residual of the returned vector. Retrieve it with ‘attr(result, "convergence")‘.

Examples

```
edges <- data.frame(
  from = c("A", "B", "C"), to = c("B", "C", "A")
)
pr_results <- compute_pagerank(edges)
print(pr_results)
if (nrow(pr_results) > 0) sum(pr_results$pagerank)

# With specified vertices (e.g., from drop_isolates)
vertices <- data.frame(
  node_name = c("A", "B", "C", "D")
) # D is an isolate
pr_results_isolates_kept <- compute_pagerank(edges, vertices_df = vertices)
print(pr_results_isolates_kept)
if (nrow(pr_results_isolates_kept) > 0) {
  sum(pr_results_isolates_kept$pagerank)
}

# Single node graph with self-loop
single_node_edges <- data.frame(
  from = "A", to = "A"
)
compute_pagerank(single_node_edges)

# Single node, no edges, defined by vertices_df
single_node_no_loop <- data.frame(
  from = character(0), to = character(0)
)
compute_pagerank(
  single_node_no_loop,
  vertices_df = data.frame(node_name = "A")
)

# Empty graph (no edges, no vertices defined)
```

```
empty_edges <- data.frame(
  from = character(), to = character()
)
compute_pagerank(empty_edges)

# Edges with NAs (these edges will be dropped)
edges_with_na <- data.frame(
  from = c("A", NA, "C"), to = c("B", "D", NA)
)
compute_pagerank(edges_with_na) # Should only process A->B
compute_pagerank(
  edges_with_na,
  vertices_df = data.frame(node_name = c("A", "B", "C", "D"))
)
```

`compute_salsa` *Compute SALSA hub and authority scores*

Description

Computes the Stochastic Approach for Link-Structure Analysis (SALSA; Lempel & Moran 2001) hub and authority scores from a processed edge list. SALSA combines HITS-style mutual reinforcement with PageRank-style stochastic random walks on the bipartite hub/authority graph. This is the low-level computational core; the high-level [salsa()] wrapper runs the URL-cleaning, redirect/canonical folding, domain filtering, deduplication, and isolate-handling identity pipeline first.

Usage

```
compute_salsa(
  edge_list_df,
  vertices_df = NULL,
  from_col = "from",
  to_col = "to",
  vertex_col_name = "node_name",
  pr_node_col = "node_name",
  hub_col = "hub",
  authority_col = "authority"
)
```

Arguments

`edge_list_df`	A data frame representing the processed edge list, with source/target columns (see ‘from_col‘, ‘to_col‘). NAs in those columns are omitted before graph construction.
`vertices_df`	An optional single-column data frame of node names defining the vertex set (e.g. to retain isolates). If ‘NULL‘ (default), the vertices are inferred from ‘edge_list_df‘. The column name is given by ‘vertex_col_name‘.

`from_col`, `to_col`
: Names of the source/target columns in ‘edge_list_df‘. Defaults ‘"from"‘ / ‘"to"‘.

`vertex_col_name`
: Name of the node column in ‘vertices_df‘. Default ‘"node_name"‘.

`pr_node_col`
: Name for the node column in the output. Default ‘"node_name"‘ (kept consistent with [compute_pagerank()]).

`hub_col`, `authority_col`
: Names for the hub and authority score columns in the output. Defaults ‘"hub"‘ / ‘"authority"‘.

Details

The two SALSA Markov chains

SALSA builds an undirected bipartite graph $\hat{G}$: each crawl-graph edge $u \to v$ contributes a hub-node u_h and an authority-node v_a joined by an edge. Two coupled random walks run on it. The **authority** chain alternates authority → hub → authority (one step = two traversals); the **hub** chain alternates the other way. Unlike HITS — whose scores are the dominant eigenvectors of $A^\top A$ and $AA^\top$ — each SALSA chain is *stochastic*, so its stationary distribution is the score vector.

Closed form (no iteration)

Lempel & Moran (2001, Proposition 6) show the stationary distributions have a degree-based closed form, so **no eigenvector iteration is needed**. On a single connected component the authority score of a node is $d_{in}(i)/W$ and the hub score is $d_{out}(i)/W$, where W is the edge count. When the support graph splits into several weakly connected components, each component's scores are renormalized within the component and then reweighted by the component's share of the relevant side (Proposition 6):

$$\tilde{\pi}_j = \frac{|A_{c(j)}|}{|A|} \times \frac{d_{in}(j)}{W_{c(j)}}$$

for authorities (and symmetrically for hubs with d_{out} and $|H_c|$), where A is the set of all authorities (in-degree > 0), $A_{c(j)}$ the authorities in j's component, and $W_{c(j)}$ the edges in that component. **This component reweighting is required for correctness:** without it, cross-component score comparisons are invalid — a common failure mode on site crawls with orphan page clusters. Each side's scores sum to ‘1‘.

Coverage and one-sided vertices

The hub side contains only nodes with out-degree > 0; the authority side only nodes with in-degree > 0. A node's ‘hub‘ is ‘NA‘ when its out-degree is ‘0‘, and its ‘authority‘ is ‘NA‘ when its in-degree is ‘0‘ (a pure sink has ‘NA‘ hub; a pure source has ‘NA‘ authority; an isolate has both ‘NA‘). SALSA coverage therefore differs from PageRank coverage on the same graph — this is expected, not a bug.

Weighting

v1 is **unweighted**: the closed form assumes uniform edge weights, so scores are driven by in-/out-degree on the deduplicated simple graph. A weighted extension is deferred.

Value

A data frame with three columns: the node name (named by ‘pr_node_col‘) and the hub and authority scores (named by ‘hub_col‘ / ‘authority_col‘). Hub and authority each sum to ‘1‘ over their non-‘NA‘ entries. Returns an empty (zero-row) data frame with those columns when the graph has no vertices.

References

Lempel, R. & Moran, S. (2001). SALSA: The Stochastic Approach for Link-Structure Analysis. *ACM Transactions on Information Systems*, 19(2), 131-160.

See Also

[salsa()] for the full identity pipeline; [compute_hits()] for the HITS analogue; [compute_pagerank()] for the PageRank analogue.

Examples

```
edges <- data.frame(
  from = c("A", "A", "B"), to = c("B", "C", "C")
)
compute_salsa(edges)

# Retain an isolate via vertices_df (NA hub and NA authority)
verts <- data.frame(node_name = c("A", "B", "C", "D"))
compute_salsa(edges, vertices_df = verts)
```

---

damping_sensitivity *Sweep PageRank across a range of damping factors*

Description

Runs [pagerank()] at each damping factor α in ‘alphas‘ and returns a tidy data frame of per-URL scores alongside the convergence metadata for each solve. This makes the sensitivity of the ranking to α directly inspectable on *your* graph, rather than relying on the field default of ‘0.85‘ (see the "Damping factor" section of [pagerank()] for why that default is only an empirical convention).

Usage

```
damping_sensitivity(edge_list_df, alphas = c(0.75, 0.8, 0.85, 0.9, 0.95), ...)
```

Arguments

edge_list_df	A data frame representing the edge list, passed to every [pagerank()] call. (Named for consistency with the rest of the package; it is an edge list, not a constructed graph object.)
alphas	Numeric vector of damping factors to sweep, each strictly between 0 and 1. Default ‘c(0.75, 0.80, 0.85, 0.90, 0.95)‘. Duplicate values are dropped.

... Additional arguments forwarded to [pagerank()] (e.g. ‘redirects_df‘, ‘weight_col‘, ‘algo‘, ‘eps‘, ‘niter‘, ‘prior_df‘). Passing ‘damping‘ here is an error, since ‘alphas‘ is what drives the damping factor.

Details

The helper is the empirical companion to the closed-form α-derivative analysis of Boldi, Santini & Vigna (*PageRank as a Function of the Damping Factor*, WWW 2005): instead of differentiating the PageRank vector with respect to α analytically, it samples the vector at a grid of α values so you can see how much each page's score (and the overall ranking) actually moves. Pair it with [compare_pagerank()] to quantify the rank churn between any two α values.

Each row also carries the convergence metadata for that α's solve. The empirical ‘iters‘ count is only reported by the ARPACK solver; under the default PRPACK direct solver it is ‘NA‘ (PRPACK exposes no iteration count). To populate it, forward ‘algo = "arpack"‘ (or an ‘eps‘ / ‘niter‘ control) through ‘...‘. The solver-independent ‘iters_estimate‘ column is always populated: it is the power-iteration rule of thumb $\lceil \log_{10}(\tau) / \log_{10}(\alpha) \rceil$ (Langville & Meyer, 2004) at the convergence tolerance τ, and shows how the required iteration count climbs as α approaches 1 regardless of solver.

Value

A tidy data frame with one row per (URL, α) pair, sorted by ‘alpha‘ ascending then ‘score‘ descending, with columns:

‘url‘ Node / page identifier.

‘alpha‘ The damping factor used for this solve.

‘score‘ The page's PageRank score at this ‘alpha‘.

‘iters‘ Iterations the solver used (ARPACK only; ‘NA‘ under PRPACK).

‘iters_estimate‘ Power-iteration iteration-count estimate at the convergence tolerance (solver-independent).

‘residual‘ Post-hoc L1 residual $\|Gx - x\|_1$ of the solve.

‘converged‘ Whether the residual met the tolerance.

A ‘"convergence"‘ attribute is attached: a compact one-row-per-‘alpha‘ data frame (‘alpha‘, ‘algo‘, ‘iters‘, ‘iters_estimate‘, ‘residual‘, ‘tol‘, ‘converged‘, ‘n_nodes‘) summarizing each solve.

See Also

[pagerank()] (the "Damping factor" section), [pagerank_convergence], [compare_pagerank()], [pagerank_grid()]

Examples

```
edges <- data.frame(
  from = c("A", "B", "C", "A", "D"),
  to = c("B", "C", "A", "C", "A")
)
sens <- damping_sensitivity(edges, clean_edge_urls = FALSE)
print(sens)
```

```
attr(sens, "convergence")

# Populate the empirical iteration count by using the ARPACK solver.
sens_ar <- suppressMessages(
  damping_sensitivity(edges, algo = "arpack", clean_edge_urls = FALSE)
)
attr(sens_ar, "convergence")
```

`drop_isolates` *Identify and Optionally Drop Isolated Nodes from an Edge List*

Description

From an edge list, identifies isolated nodes (nodes that do not participate in any complete edge, i.e., a row where both from and to are non-NA). It can return only connected nodes (degree > 0) or the full vertex universe (all unique non-NA URLs from both columns, including those from partial/incomplete rows).

Usage

```
drop_isolates(
  edge_list_df,
  drop = FALSE,
  from_col = "from",
  to_col = "to",
  node_col_name = "node_name"
)
```

Arguments

`edge_list_df`	A data frame representing the edge list, typically with columns "from" and "to" (or as specified by ‘from_col‘, ‘to_col‘). Rows where both columns are non-NA represent edges. Rows where one column is NA represent known nodes that do not participate in a complete edge (potential isolates).
`drop`	Logical. If ‘TRUE‘, returns a single-column data frame containing only node names that participate in at least one complete edge (both from and to are non-NA in the same row). If ‘FALSE‘ (default), returns a single-column data frame of all unique non-NA node names present in either column of ‘edge_list_df‘ (the full vertex universe, including isolates).
`from_col`	Name of the source node column in ‘edge_list_df‘. Default "from".
`to_col`	Name of the target node column in ‘edge_list_df‘. Default "to".
`node_col_name`	Name for the output column containing node names. Default "node_name". When used with [compute_pagerank()], this should match its ‘vertex_col_name‘ parameter.

Value

A single-column data frame named according to ‘node_col_name‘. If ‘drop = TRUE‘, contains unique node names with degree > 0 (from complete edges only). If ‘drop = FALSE‘, contains all unique non-NA node names from both columns of the input edge list (full vertex universe). Returns an empty data frame with the correct column name if no nodes meet the criteria or if the input edge list is empty/all NAs.

Examples

```
# Edge list with partial rows
# (NA in one column = known node, not a complete edge)
edges <- data.frame(
  from = c("A", "B", "C", NA, "D"),
  to = c("B", "C", "A", "E", NA)
)
# Complete edges: A->B, B->C, C->A.
# Partial rows: NA->E (E is isolate), D->NA (D is isolate).

# Get only nodes participating in complete edges (A, B, C)
active_nodes <- drop_isolates(edges, drop = TRUE)
print(active_nodes)

# Get all unique nodes including isolates from partial rows (A, B, C, D, E)
all_nodes <- drop_isolates(edges, drop = FALSE)
print(all_nodes)

# Edge list with no isolates (all rows are complete edges)
edges_complete <- data.frame(
  from = c("X", "Y"),
  to = c("Y", "X")
)
drop_isolates(edges_complete, drop = TRUE) # X, Y
drop_isolates(edges_complete, drop = FALSE) # X, Y (same, no partial rows)

# Empty edge list
empty_edges <- data.frame(
  from = character(0), to = character(0)
)
drop_isolates(empty_edges, drop = TRUE)
drop_isolates(empty_edges, drop = FALSE)

# Edge list with only NAs
na_edges <- data.frame(
  from = NA_character_, to = NA_character_
)
drop_isolates(na_edges, drop = TRUE)
drop_isolates(na_edges, drop = FALSE)

# Custom column names
custom_edges <- data.frame(
  source = c("S1"), target = c("T1")
)
```

```
drop_isolates(
  custom_edges,
  from_col = "source", to_col = "target", node_col_name = "vertex"
)
```

`export_graph` *Export PageRank Graph*

Description

Exports a PageRank result and its edge list as a graph file suitable for visualization in external tools (Gephi, yEd, Graphviz, etc.). Supports GraphML, GEXF (via GraphML with attributes), DOT, and edge list CSV formats.

Usage

```
export_graph(
  pagerank_df,
  edge_list_df,
  file,
  format = c("graphml", "dot", "edgelist", "pajek"),
  edge_from_col = "from",
  edge_to_col = "to",
  pr_url_col = "node_name",
  pr_score_col = "pagerank",
  node_attrs = NULL,
  edge_attrs = NULL
)
```

Arguments

`pagerank_df`	A data frame with at least `url` and `pagerank` columns, as returned by `pagerank`.
`edge_list_df`	A data frame of edges with from/to columns.
`file`	Character, path to the output file.
`format`	Character, output format. One of `"graphml"`, `"dot"`, `"edgelist"`, or `"pajek"`.
`edge_from_col, edge_to_col`	Names of from/to columns in `edge_list_df`. Default `"from"` and `"to"`.
`pr_url_col`	Name of the URL column in `pagerank_df`. Default `"node_name"` (matching `pagerank` output).
`pr_score_col`	Name of the PageRank score column. Default `"pagerank"`.
`node_attrs`	Optional named list of additional vertex attribute columns from `pagerank_df` to include (e.g., `list(rank = "rank")`).
`edge_attrs`	Optional character vector of additional columns from `edge_list_df` to include as edge attributes (e.g., `"weight"`).

Value

The file path (invisibly). Called for its side effect of writing a file.

Examples

```
edges <- data.frame(
  from = c("A", "B", "C"),
  to = c("B", "C", "A")
)
pr <- pagerank(edges, clean_edge_urls = FALSE)

# Export to GraphML (for Gephi)
tmp <- tempfile(fileext = ".graphml")
export_graph(pr, edges, file = tmp, format = "graphml")

# Export as DOT (for Graphviz)
tmp_dot <- tempfile(fileext = ".dot")
export_graph(pr, edges, file = tmp_dot, format = "dot")
```

---

filter_links_by_domain

Filter Edge List by Domain or Host

Description

Filters an edge list by registrable domain and/or host rules. Rows are kept only when both endpoints satisfy the keep/ignore logic. Ignore rules always override keep rules. When keep rules are provided, ‘drop_third_party = TRUE‘ removes URLs outside the keep lists.

This function is intended as a **pre-processing step** before calling [pagerank()]. For example, to scope a PageRank analysis to a single site or exclude CDN / tracking domains.

Ordering relative to folding: when ‘pagerank()‘ calls this filter internally (via its ‘keep_domains‘ / ‘exclude_domains‘ / ‘keep_hosts‘ / ‘exclude_hosts‘ arguments), the filter runs *after* redirect and canonical folding, so it scopes the post-fold (canonical) namespace. If an out-of-scope canonical/redirect rewrites the crawled domain/host onto a different one, filtering on the crawled value inside ‘pagerank()‘ matches nothing. To domain-scope the **crawled input** instead, call ‘filter_links_by_domain()‘ on the edge list yourself *before* folding (i.e. before passing it to ‘pagerank()‘).

Usage

```
filter_links_by_domain(
  edge_list_df,
  from_col = "from",
  to_col = "to",
  keep_domains = NULL,
  keep_hosts = NULL,
  ignore_domains = NULL,
```

```
  ignore_hosts = NULL,
  drop_third_party = TRUE,
  return_report = FALSE,
  psl_section = c("all", "icann", "private"),
  rurl_params = list()
)
```

Arguments

edge_list_df	A data frame representing the edge list, with at least two URL columns.
from_col	Name of the source URL column. Default ‘"from"‘.
to_col	Name of the target URL column. Default ‘"to"‘.
keep_domains	Character vector of registrable domains to keep (e.g., ‘"example.com"‘). Subdomains are included when their registrable domain matches.
keep_hosts	Character vector of specific hosts to keep (e.g., ‘"www.example.com"‘). Only exact host matches are kept.
ignore_domains	Character vector of registrable domains to drop.
ignore_hosts	Character vector of specific hosts to drop.
drop_third_party	Logical. When keep lists are provided and this is ‘TRUE‘ (default), URLs outside the keep lists are dropped. When ‘FALSE‘, only explicitly ignored URLs are dropped.
return_report	Logical. If ‘TRUE‘, returns a list with the filtered data frame and a filter report. Default ‘FALSE‘.
psl_section	Public Suffix List section used to derive registrable domains, passed to ‘rurl::get_domain()‘ / ‘rurl::safe_parse_urls()‘. One of ‘"all"‘ (default, ICANN + private suffixes), ‘"icann"‘, or ‘"private"‘. Affects domain-based (not host-based) keep/ignore matching; e.g. under ‘"icann"‘, ‘user.github.io‘ has registrable domain ‘github.io‘, while under ‘"all"‘ it is ‘user.github.io‘.
rurl_params	A list of ‘rurl‘ canonicalization arguments overriding pagerankr’s profile per key, used when extracting hosts/domains from both the edge URLs and the keep/ignore values. Pass the **same** profile used to clean the graph so the comparison keys are derived identically. The host-relevant knobs are ‘host_encoding‘ (‘"keep"‘/‘"idna"‘/‘"unicode"‘ — IDN folding; e.g. ‘"idna"‘ makes ‘münchen.de‘ and ‘xn–mnchen-3ya.de‘ match), ‘www_handling‘, ‘subdomain_levels_to_keep‘, ‘case_handling‘, and ‘protocol_handling‘. Registrable-domain matching is encoding-independent. When called from [pagerank()], this is forwarded automatically.

Value

If ‘return_report = FALSE‘ (default), the filtered data frame (preserving all columns). If ‘TRUE‘, a list with elements ‘filtered_df‘ and ‘report‘.

Examples

```
links <- data.frame(
  from = c(
    "http://www.example.com/a", "http://example.com/b",
    "http://cdn.tracker.com/c"
  ),
  to = c(
    "http://example.com/b", "http://help.example.com/d",
    "http://www.example.com/a"
  )
)

# Keep only example.com edges
filter_links_by_domain(links, keep_domains = "example.com")

# Ignore a specific subdomain
filter_links_by_domain(links, ignore_hosts = "cdn.tracker.com")

# Get a report of what was filtered
result <- filter_links_by_domain(links,
  keep_domains = "example.com",
  return_report = TRUE
)
result$report
```

---

ga4_entrance_teleport *GA4 Entrance / Landing-Page Teleport Adapter (PROXY)*

Description

Turns GA4 **entrance / landing-page counts** into the teleport (reset / personalization) vector for weighted PageRank with an *entrance-biased reset*. Each session start is treated as a teleport event whose destination is the landing page, so pages where users more often *begin* a session receive proportionally more of the random surfer's reset mass — replacing the uniform teleport of standard PageRank.

This is the cheapest of the three behavioral-reset models (it reuses the standard PageRank machine unchanged), and it is deliberately a **proxy**: see the dedicated note below.

Usage

```
ga4_entrance_teleport(
  entrances_df,
  url_col = "url",
  entrances_col = "entrances",
  vertex_names = NULL,
  transform = c("none", "log", "percentile", "minmax", "zipf", "rank_linear"),
  alpha = 0,
  exclude_nodes = character(0),
```

```
  verbose = TRUE
)
```

Arguments

entrances_df	A data frame with one row per (landing page, entrance count) observation, e.g. a GA4 "Landing page" report. Multiple rows for the same URL are summed (entrances are additive raw counts). Rows with a missing URL or a missing / negative count are dropped.
url_col	Name of the landing-page URL column in entrances_df. Default "url".
entrances_col	Name of the numeric entrance-count column in entrances_df. Default "entrances". **Contract: this must be an additive raw count** (session starts / entrances), never a rate, share, or computed score — folding two URLs onto one representative is a meaningful sum. This mirrors the TIPR additive-count contract in [align_prior_to_vertices()].
vertex_names	Optional character vector of the graph's vertex names, in graph order (e.g. igraph::V(graph)$name). When supplied, the adapter returns the **aligned teleport vector** directly by delegating to [align_prior_to_vertices()] (passing transform, alpha, exclude_nodes, verbose). The vertex_names you pass must already be canonicalized + redirect-folded like the edges; pass NULL (the default) to get back a prior_df-shaped data frame and let [pagerank()] perform that canonicalization + fold (the recommended path — single source of truth for folding).
transform, alpha, exclude_nodes, verbose	Passed through to [align_prior_to_vertices()] when vertex_names is supplied; ignored otherwise. See that function for semantics. alpha = 1 recovers the standard uniform teleport.

Details

Uniform entrances recover uniform teleport. If every (real) vertex has the same entrance count, the entrance share is uniform, so the resulting teleport vector equals the standard uniform PageRank reset — the proxy degrades gracefully to the default when there is no entrance signal.

Recommended usage (let pagerank() own the fold):

```
tp <- ga4_entrance_teleport(ga4_landing_report,
                            url_col = "landing_page",
                            entrances_col = "sessions")
pagerank(edges, prior_df = tp)   # prior_df = data.frame(url, weight)
```

Value

If vertex_names is NULL (default), a data frame with columns url and weight (one row per unique landing-page URL, entrances summed) ready to pass to pagerank(prior_df = , alpha =). If vertex_names is supplied, a numeric teleport vector the same length and order as vertex_names, summing to 1 (the return value of [align_prior_to_vertices()]).

This is a PROXY, not an identity

Session starts are **not literally equivalent to every PageRank teleport event**. The teleport in PageRank fires on *every* damping draw (including mid-session "I got bored, jump elsewhere" restarts), while GA4 entrances only observe the *first* page of a session. Using entrances as the reset distribution is a defensible approximation of "where browsing tends to (re)start," but it is an approximation. Higher-fidelity models — page-specific exit probabilities (a discrete behavioral Markov model) and continuous-time BrowseRank with dwell time — are explicitly **out of scope here**. Treat, report, and cite this vector as the *entrance-biased teleport proxy*.

Distinct from the backlink-authority prior

This adapter and the external-authority TIPR prior (e.g. Ahrefs referring domains; see [align_prior_to_vertices()]) both flow through the same `prior_df` / [align_prior_to_vertices()] plumbing, but they answer **different questions** and should not be conflated:

- **Backlink-authority prior** — where *authority enters* the graph from outside (off-site links). A structural/link signal.
- **Entrance teleport (this adapter)** — where *users* enter / restart browsing (observed session starts). A behavioral signal, and only a proxy for the teleport event (see above).

They can be used as alternatives, or — outside this function's remit — blended; that mixing policy is not decided here.

Naming decision (prior_df vs teleport_df/reset_df)

We **reuse the existing** `prior_df` **machinery** rather than introducing a separate `teleport_df` / `reset_df`. Rationale: (1) entrances are **additive raw counts**, so they satisfy the same TIPR additive-count contract as referring-domain counts — duplicate / redirect-folded URLs combine by summation, which is exactly what [align_prior_to_vertices()] already does; (2) both signals produce a teleport vector over the *same* final vertex set with the *same* canonicalization + redirect fold, so a parallel data-frame type and a parallel alignment path would be duplicated machinery for no behavioral gain; (3) the *semantic* distinction (authority-in vs users-in) is carried by documentation and by the proxy labeling here, not by the data structure. If a future model needs to *blend* a backlink prior and an entrance reset in a single `pagerank()` call, that is the point to revisit and split the type (tracked as research-notes Q5 / Q3).

See Also

[align_prior_to_vertices()], [pagerank()], [transform_weights()]

Examples

```
ga4 <- data.frame(
  url = c("https://x/a", "https://x/a", "https://x/b"),
  entrances = c(60, 30, 10)
)
# As a prior_df for pagerank() (it does the canonicalize + fold):
ga4_entrance_teleport(ga4)

# Or align directly to a known final vertex set:
```

```
v <- c("https://x/a", "https://x/b", "https://x/c")
ga4_entrance_teleport(ga4, vertex_names = v, verbose = FALSE)
```

`ga4_page_transitions` *Build Page-Transition Counts from a GA4 BigQuery Export*

Description

Builds consecutive-page-view **transition counts** from a Google Analytics 4 (GA4) BigQuery event-export data frame. The result is a ‘from‘/‘to‘ edge list with a count column, in the shape that [pagerank()] accepts (pass the count column via ‘weight_col‘).

This function operates entirely on a data frame **you supply** — it does **not** query BigQuery and adds no database dependencies. Extract the GA4 ‘events_*‘ rows you care about (typically ‘page_view‘ events, with the session-identity and ordering fields un-nested from ‘event_params‘ / the ‘batch‘ struct) into a data frame, then pass it here.

Usage

```
ga4_page_transitions(
  events_df,
  user_id_col = "user_pseudo_id",
  session_id_col = "ga_session_id",
  page_col = "page_location",
  timestamp_col = "event_timestamp",
  batch_page_id_col = "batch_page_id",
  batch_ordering_id_col = "batch_ordering_id",
  batch_event_index_col = "batch_event_index",
  from_col = "from",
  to_col = "to",
  count_col = "n",
  drop_self_transitions = TRUE
)
```

Arguments

`events_df`	A data frame of GA4 export rows, one row per event (typically filtered to ‘page_view‘ events upstream). Must contain the session-identity, page, and timestamp columns named below; the ‘batch_*‘ tie-break columns are optional but recommended.
`user_id_col`	Name of the user-identity column. GA4 default ‘"user_pseudo_id"‘.
`session_id_col`	Name of the session-identity column (the un-nested ‘ga_session_id‘ event parameter). GA4 default ‘"ga_session_id"‘.
`page_col`	Name of the page-identity column whose consecutive values form the transitions. GA4 default ‘"page_location"‘.
`timestamp_col`	Name of the primary ordering column. GA4 default ‘"event_timestamp"‘.

`batch_page_id_col`, `batch_ordering_id_col`, `batch_event_index_col`
: Names of the GA4 batch tie-break columns, applied in this order after ‘timestamp_col‘. GA4 defaults ‘"batch_page_id"‘, ‘"batch_ordering_id"‘, ‘"batch_event_index"‘. A column that is not present in ‘events_df‘ is skipped.

`from_col`, `to_col`
: Names of the source/target columns in the returned edge list. Defaults ‘"from"‘ / ‘"to"‘ (the [pagerank()] defaults).

`count_col`
: Name of the transition-count column in the returned edge list. Default ‘"n"‘. Pass this name to ‘pagerank(weight_col = ...)‘.

`drop_self_transitions`
: Logical. If ‘TRUE‘ (default), consecutive page views of the **same** page (reloads, SPA re-renders to the same route) are dropped before counting. If ‘FALSE‘, self-transitions are kept and counted.

Value

A data frame with one row per distinct ‘from -> to‘ page transition, carrying the columns named by ‘from_col‘, ‘to_col‘, and ‘count_col‘. The count column is an integer tally of how many times that consecutive page-view transition was observed across all sessions. Rows are ordered by ‘from‘ then ‘to‘ for stable output. When no transitions exist (e.g. every session has a single page view), an empty data frame with the correct columns is returned.

What this measures (transition, NOT link-click)

The output is a **behavioral navigation signal** — the empirical "where did users go next" sequence of page views within a session. It is **not** a measured link-click probability. GA4 page-view sequences are contaminated by page reloads, browser back/forward navigation, server redirects, single-page-application route changes, dropped/missing events, and off-site returns. A transition ‘A -> B‘ means "a session viewed page A and then viewed page B next", which is **not** the same as "a user clicked a link from A to B." For link-click instrumentation (the actual element clicked), a separate ‘ga4_link_clicks()‘ product is required; do not use this function as a substitute for it.

Session / event ordering contract

Within each session, events are ordered by ‘event_timestamp‘ and then by a deterministic chain of tie-break fields, **in this order**:

1. ‘event_timestamp‘ (microseconds since epoch),
2. ‘batch_page_id‘,
3. ‘batch_ordering_id‘,
4. ‘batch_event_index‘.

‘event_timestamp‘ **alone is insufficient**: GA4 batches events and multiple events in a session can share the exact same ‘event_timestamp‘. When timestamps tie, the ‘batch_*‘ fields (assigned by the GA4 SDK in the order events were recorded on the client) break the tie so the ordering is stable and reproducible. Any tie-break column that is absent from ‘events_df‘ is simply skipped, but supplying all of them is strongly recommended to guarantee a deterministic order. As a final

stabilizer the original row order of ‘events_df‘ is used, so the result never depends on the platform’s sort implementation.

A *session* is identified by the combination of ‘user_id_col‘ and ‘session_id_col‘ (GA4: ‘user_pseudo_id‘ and the ‘ga_session_id‘ event parameter). Transitions are only formed **within** a single session; consecutive page views that cross a session boundary are never joined.

See Also

[pagerank()] for consuming the result; [transform_weights()] for turning raw transition counts into PageRank edge weights.

Examples

```
events <- data.frame(
  user_pseudo_id = c("u1", "u1", "u1", "u2", "u2"),
  ga_session_id = c(1, 1, 1, 9, 9),
  page_location = c("/home", "/blog", "/contact", "/home", "/blog"),
  event_timestamp = c(100, 200, 300, 100, 200),
  batch_page_id = c(0, 1, 2, 0, 1),
  batch_ordering_id = c(0, 0, 0, 0, 0),
  batch_event_index = c(0, 1, 2, 0, 1)
)
transitions <- ga4_page_transitions(events)
transitions
# Feed to pagerank() as a behavioral transition model:
# pagerank(transitions, weight_col = "n", clean_edge_urls = FALSE)
```

---

get_unique_edges *Get Unique Edges from an Edge List*

Description

Removes duplicate edge rows from an edge list data frame and provides control over how self-loops (e.g., a -> a) are handled.

Usage

```
get_unique_edges(
  edge_list_df,
  self_loops = c("drop", "keep"),
  from_col = "from",
  to_col = "to"
)
```

Arguments

edge_list_df	A data frame representing the edge list, typically with columns "from" and "to" (or as specified by ‘from_col‘, ‘to_col‘).
self_loops	A character string specifying how to handle self-loops. Must be one of "drop" (default) or "keep".
from_col	Name of the source node column in ‘edge_list_df‘. Default "from".
to_col	Name of the target node column in ‘edge_list_df‘. Default "to".

Details

Any edge where either from or to is NA is automatically dropped before deduplication and self-loop handling.

Value

A data frame with unique edges, with self-loops handled according to the ‘self_loops‘ argument. The from/to columns are coerced to character; all other columns in the input are preserved (first occurrence kept on dedup). If input columns are factors, they are converted to characters in the output.

Examples

```
edges <- data.frame(
  from = c("A", "B", "A", "C", "D"),
  to = c("B", "C", "B", "C", "D")
)
get_unique_edges(edges, self_loops = "drop")
get_unique_edges(edges, self_loops = "keep")

# With custom column names
edges_custom <- data.frame(
  source = c("X", "Y", "X"),
  target = c("Y", "Y", "Y")
)
get_unique_edges(edges_custom, from_col = "source", to_col = "target")

# With NAs (NAs are preserved as they are,
# duplicates involving NAs are also removed)
edges_na <- data.frame(
  from = c("A", NA, "A", "B", NA),
  to = c("B", "C", "B", "D", "C")
)
get_unique_edges(edges_na, self_loops = "keep")
# No self-loops with NA to drop
get_unique_edges(edges_na, self_loops = "drop")
```

hits *Master HITS hub/authority calculation wrapper*

Description

Computes Kleinberg's HITS hub and authority scores over the same cleaned, redirect/canonical-folded, domain-filtered, deduplicated link graph that [pagerank()] builds, so node identities line up across the two centrality measures. Wraps 'igraph::hits_scores()' (the non-deprecated successor of 'igraph::hub_score()' / 'igraph::authority_score()').

Usage

```
hits(
  edge_list_df,
  redirects_df = NULL,
  clean_edge_urls = TRUE,
  clean_redirect_urls = TRUE,
  rurl_params = list(),
  self_loops = c("drop", "keep"),
  drop_isolates_flag = TRUE,
  weight_col = NULL,
  duplicate_edge_policy = c("collapse", "aggregate", "count_instances"),
  edge_from_col = "from",
  edge_to_col = "to",
  redirect_from_col = "from",
  redirect_to_col = "to",
 duplicate_from_policy = c("strict", "first_wins", "last_wins", "most_frequent",
    "prune_source", "resolve_if_consistent"),
  loop_handling = c("error", "prune_loop", "break_arrow"),
  canonicals_df = NULL,
  canonical_from_col = "from",
  canonical_to_col = "to",
  clean_canonical_urls = TRUE,
  canonical_duplicate_from_policy = c("strict", "first_wins", "last_wins",
    "most_frequent", "prune_source", "resolve_if_consistent"),
  canonical_loop_handling = c("error", "prune_loop", "break_arrow"),
  canonical_conflict_policy = c("redirect_wins", "error", "canonical_wins"),
  keep_domains = NULL,
  exclude_domains = NULL,
  keep_hosts = NULL,
  exclude_hosts = NULL,
  scale = TRUE,
  ...
)
```

Arguments

edge_list_df	A data frame representing the edge list, typically with columns like "from" and "to". Edges are expected to be page-to-page hyperlinks: ‘pagerank()‘ is graph-agnostic and treats every endpoint as a node, so resource links (images, CSS, JS, and other non-HTML references) must be filtered upstream or they collect authority as ordinary vertices. ‘pagerank_screaming_frog()‘ does this at the crawl boundary via [sf_graph_eligible()] (‘Hyperlink‘ only); a hand-built or non-SF edge list should apply the same hyperlink-only filter before scoring.
redirects_df	An optional data frame for redirect rules, typically with "from" and "to" columns. Defaults to NULL.
clean_edge_urls	Logical, whether to clean URLs in the edge list. Defaults to TRUE.
clean_redirect_urls	Logical, whether to clean URLs in the redirect list. Defaults to TRUE. Only effective if ‘redirects_df‘ is provided.
rurl_params	A list of parameters to pass to ‘rurl::clean_url‘. Defaults to an empty list. ‘protocol_handling‘ defaults to ‘"keep"‘ and ‘case_handling‘ to ‘"lower_host"‘ for cross-project canonicalization consistency. If you set ‘host_encoding‘ (‘"idna"‘ or ‘"unicode"‘) to fold internationalized (IDN) hosts, that same value is also passed to the domain-filtering step so its comparisons stay consistent with the cleaned node keys. (Registrable-domain matching is encoding-independent, so this only matters if host-level filtering is involved.)
self_loops	A character string specifying how to handle self-loops. Either "drop" (default) or "keep".
drop_isolates_flag	Logical, whether to drop isolated nodes before PageRank computation. Defaults to TRUE.
weight_col	Optional name of a numeric column in ‘edge_list_df‘ containing edge weights. Higher weights make edges more likely to be followed. If ‘NULL‘ (default), all edges have equal weight.
duplicate_edge_policy	How repeated ‘from -> to‘ rows are represented after URL cleaning, redirect/canonical folding, and domain filtering. One of: **‘"collapse"‘** (default) Destination-level surfer: repeated rows collapse to one unweighted destination edge, preserving legacy ‘get_unique_edges()‘ behavior and the common binary PageRank convention. **‘"aggregate"‘** Collapse each ‘from -> to‘ pair with [aggregate_edges()] semantics. Numeric columns, including ‘weight_col‘, are summed; logical columns such as ‘nofollow‘ use the default ‘"any"‘ conflict policy. **‘"count_instances"‘** Link-slot / edge-level surfer: repeated rows increase transition probability. With no ‘weight_col‘, each surviving ‘from -> to‘ pair receives an internal weight equal to its duplicate-row count. With ‘weight_col‘, weights are summed and an ‘instance_count‘ audit column is retained.
edge_from_col, edge_to_col	Names of from/to columns in ‘edge_list_df‘.

redirect_from_col, redirect_to_col
: Names of from/to columns in ‘redirects_df‘.

duplicate_from_policy
: How to handle conflicting redirects in ‘redirects_df‘. Passed through to [resolve_redirects()]. Default ‘"strict"‘ (error on conflicts). See [resolve_redirects()] for all available policies.

loop_handling
: How to handle redirect cycles. Passed through to [resolve_redirects()]. Default ‘"error"‘. See [resolve_redirects()] for all available policies.

canonicals_df
: An optional data frame of declared ‘rel=canonical‘ links, with ‘from‘/‘to‘ columns (or as set by ‘canonical_from_col‘ / ‘canonical_to_col‘) pairing a source URL with the canonical it declares. Default ‘NULL‘ (opt-in; the default preserves current behavior). Canonicals are a **distinct, advisory** signal from enforced 3xx ‘redirects_df‘: they are tracked separately and audited via [audit_canonicals()] / [audit_fold()], then folded into the same composed map as redirects (see [build_fold_map()]). Self-canonicals drop as no-ops.

canonical_from_col, canonical_to_col
: From/to columns in ‘canonicals_df‘. Default ‘"from"‘ / ‘"to"‘.

clean_canonical_urls
: Logical, whether to clean URLs in ‘canonicals_df‘ using the same resolved ‘rurl_params‘ profile as edge and redirect cleaning. Default ‘TRUE‘. Only effective when ‘canonicals_df‘ is provided.

canonical_duplicate_from_policy
: How to handle a canonical source that declares multiple distinct canonicals. Reuses the ‘duplicate_from_policy‘ enum (see [resolve_redirects()]). Default ‘"strict"‘.

canonical_loop_handling
: How to handle cycles among declared canonicals. Reuses the ‘loop_handling‘ enum. Default ‘"error"‘.

canonical_conflict_policy
: How to resolve a redirect-vs-canonical disagreement on the **same source** URL. One of ‘"redirect_wins"‘ (default; the 3xx wins and the canonical on a redirecting source is ignored and flagged), ‘"error"‘ (error on genuine disagreement), or ‘"canonical_wins"‘ (the declared canonical wins for that source, still flagged). See [build_fold_map()].

keep_domains
: Optional character vector of domains to keep. When provided, edges are filtered via [filter_links_by_domain()] so that only links where both endpoints belong to one of the specified domains are included. Useful for restricting to internal links. Default ‘NULL‘ (no domain filtering).

  Ordering: filtering runs *after* redirect/canonical folding, so it scopes the post-fold (canonical) namespace, not the crawled input. If an out-of-scope canonical/redirect rewrites the crawled domain onto a different one, filtering on the crawled domain matches nothing (an empty graph). To scope the INPUT you crawled, run [filter_links_by_domain()] on the edge list *before* calling ‘pagerank()‘.

exclude_domains
: Optional character vector of domains to exclude. Edges where either endpoint belongs to one of these domains are removed. Like ‘keep_domains‘, this filters

	the post-fold namespace (see the ordering note above). Default ‘NULL‘ (no exclusion).
keep_hosts	Optional character vector of exact hosts to keep (e.g. ‘"www.example.com"‘), as opposed to registrable domains. Matched on the exact host using the same canonicalization profile as cleaning, so IDN folding (‘host_encoding‘ in ‘rurl_params‘) applies consistently. Default ‘NULL‘.
exclude_hosts	Optional character vector of exact hosts to exclude. Edges where either endpoint matches one of these hosts are removed. Ignore rules override keep rules. Default ‘NULL‘.
scale	Logical, passed to ‘igraph::hits_scores()‘ via [compute_hits()]. ‘TRUE‘ (default) scales each score so its maximum is ‘1‘; ‘FALSE‘ returns the unit-norm eigenvectors. See [compute_hits()].
...	Additional arguments forwarded to [compute_hits()] and then to ‘igraph::hits_scores()‘.

Details

Relationship to the PageRank pipeline

‘hits()‘ reuses the exact identity-forming steps of [pagerank()] — URL canonicalization (the same resolved ‘rurl‘ profile), the same composed redirect + canonical fold map, the same domain/host filtering, the same ‘duplicate_edge_policy‘ deduplication, and the same self-loop / isolate handling. The resulting vertex set therefore matches ‘pagerank()‘ run with the same arguments, so hub, authority, and PageRank can be joined on ‘node_name‘ without re-canonicalizing.

The PageRank-specific, *forward-flow* modeling devices have **no HITS analogue and are intentionally not exposed**: nofollow evaporation, the indexability (noindex / robots.txt) transforms, the TIPR teleport prior, and the ‘reverse‘ flag. HITS already computes both directions of authority flow (hub is the outflow-oriented score, authority the inflow-oriented one), so a separate reversal is unnecessary.

Matrix formulation and the whole-graph caveat

With adjacency matrix A, **authority** is the dominant eigenvector of $A^\top A$ ("pages pointed to by pages that point to many things") and **hub** is the dominant eigenvector of $AA^\top$ ("pages that point to pages pointed to by many things"). See [compute_hits()].

Kleinberg's original HITS (1999) was run on a small, **query-focused base set** of pages, where the hub/authority distinction is sharply interpretable. ‘hits()‘ instead runs on the **full (or user-filtered) site graph** that ‘pagerankr‘ assembles. The eigenvector computation is identical and correct, but the interpretation shifts: scores describe hub/authority structure across the whole crawled graph rather than relevance to a specific query. Treat them as site-wide structural centralities, not query-relevance scores.

Value

A data frame with one row per node and columns ‘node_name‘, ‘hub‘, and ‘authority‘ (column names configurable via ‘...‘). Hub and authority are scaled to a maximum of ‘1‘ by default (‘scale = TRUE‘).

See Also

[compute_hits()] for the computational core, [pagerank()] for the PageRank analogue sharing this identity pipeline.

Examples

```
edges <- data.frame(
  from = c("http://A.com/", "http://A.com/", "B.com"),
  to = c("B.com", "C.com", "C.com")
)
hits(edges)

# Hub vs authority: a pure outflow page tops hub, a pure inflow page tops
# authority.
h <- hits(edges)
h[which.max(h$hub), ]
h[which.max(h$authority), ]
```

`launch_pagerank_explorer`

Launch PageRank Explorer

Description

Opens an interactive Shiny application for exploring PageRank results. Upload CSV files for edge lists, redirects, and PageRank scores, then visualize the graph interactively, inspect distributions, audit redirects, and export in multiple formats.

Usage

```
launch_pagerank_explorer(...)
```

Arguments

`...`	Additional arguments passed to `shiny::runApp` (e.g., `port`, `host`, `launch.browser`).

Details

The app requires the `shiny` and `DT` packages. For interactive network visualization, `visNetwork` is recommended (the app falls back to a static igraph plot if visNetwork is not installed).

Install optional dependencies with:

```
install.packages(c("shiny", "DT", "visNetwork"))
```

Value

Called for its side effect (launches the app). Returns invisibly.

Examples

```
if (interactive()) {
  launch_pagerank_explorer()
}
```

---

pagerank *Master PageRank Calculation Wrapper*

Description

Orchestrates the complete PageRank calculation workflow, including URL cleaning, redirect resolution, edge deduplication, indexability handling, nofollow handling, isolate handling, and PageRank computation.

Usage

```
pagerank(
  edge_list_df,
  redirects_df = NULL,
  clean_edge_urls = TRUE,
  clean_redirect_urls = TRUE,
  rurl_params = list(),
  self_loops = c("drop", "keep"),
  drop_isolates_flag = TRUE,
  reverse = FALSE,
  weight_col = NULL,
  placement_col = NULL,
  accepted_placements = NULL,
  placement_weights = NULL,
  container_col = NULL,
  boilerplate_threshold = 0.5,
  min_container_pages = 10,
  boilerplate_weight = 0.5,
  position_col = NULL,
  position_transform = c("zipf", "rank_linear"),
  position_alpha = 1,
  position_floor = 0.01,
  duplicate_edge_policy = c("collapse", "aggregate", "count_instances"),
  nofollow_col = NULL,
  nofollow_action = c("evaporate", "drop", "keep"),
  indexability_df = NULL,
  indexability_url_col = "url",
  indexability_status_col = "indexability_status",
  status_df = NULL,
  status_url_col = "url",
  status_col = "status_code",
  robots_blocked_action = c("show", "vanish"),
```

```
  edge_from_col = "from",
  edge_to_col = "to",
  redirect_from_col = "from",
  redirect_to_col = "to",
  duplicate_from_policy = c("strict", "first_wins", "last_wins", "most_frequent",
    "prune_source", "resolve_if_consistent"),
  loop_handling = c("error", "prune_loop", "break_arrow"),
  canonicals_df = NULL,
  canonical_from_col = "from",
  canonical_to_col = "to",
  clean_canonical_urls = TRUE,
  canonical_duplicate_from_policy = c("strict", "first_wins", "last_wins",
    "most_frequent", "prune_source", "resolve_if_consistent"),
  canonical_loop_handling = c("error", "prune_loop", "break_arrow"),
  canonical_conflict_policy = c("redirect_wins", "error", "canonical_wins"),
  out_of_scope_fold = c("relabel", "keep", "leak"),
  keep_domains = NULL,
  exclude_domains = NULL,
  keep_hosts = NULL,
  exclude_hosts = NULL,
  prior_df = NULL,
  prior_url_col = "url",
  prior_weight_col = "weight",
  prior_transform = c("none", "log", "percentile", "minmax", "zipf", "rank_linear"),
  prior_alpha = 0,
  prior_inject_unmatched = FALSE,
  prior_exclude_waste = TRUE,
  prior_verbose = TRUE,
  damping = 0.85,
  ...,
  preset = NULL
)
```

Arguments

edge_list_df	A data frame representing the edge list, typically with columns like "from" and "to". Edges are expected to be page-to-page hyperlinks: ‘pagerank()‘ is graph-agnostic and treats every endpoint as a node, so resource links (images, CSS, JS, and other non-HTML references) must be filtered upstream or they collect authority as ordinary vertices. ‘pagerank_screaming_frog()‘ does this at the crawl boundary via [sf_graph_eligible()] (‘Hyperlink‘ only); a hand-built or non-SF edge list should apply the same hyperlink-only filter before scoring.
redirects_df	An optional data frame for redirect rules, typically with "from" and "to" columns. Defaults to NULL.
clean_edge_urls	Logical, whether to clean URLs in the edge list. Defaults to TRUE.
clean_redirect_urls	Logical, whether to clean URLs in the redirect list. Defaults to TRUE. Only effective if ‘redirects_df‘ is provided.

`rurl_params` A list of parameters to pass to ‘rurl::clean_url‘. Defaults to an empty list. ‘protocol_handling‘ defaults to ‘"keep"‘ and ‘case_handling‘ to ‘"lower_host"‘ for cross-project canonicalization consistency. If you set ‘host_encoding‘ (‘"idna"‘ or ‘"unicode"‘) to fold internationalized (IDN) hosts, that same value is also passed to the domain-filtering step so its comparisons stay consistent with the cleaned node keys. (Registrable-domain matching is encoding-independent, so this only matters if host-level filtering is involved.)

`self_loops` A character string specifying how to handle self-loops. Either "drop" (default) or "keep".

`drop_isolates_flag`
Logical, whether to drop isolated nodes before PageRank computation. Defaults to TRUE.

`reverse` Logical. If ‘TRUE‘, PageRank is computed on the transposed (edge-reversed) graph, yielding reverse / inverse PageRank instead of the usual inflow score. Default ‘FALSE‘. See the "Reverse / inverse PageRank" section in Details for what it measures and which other arguments are compatible.

`weight_col` Optional name of a numeric column in ‘edge_list_df‘ containing edge weights. Higher weights make edges more likely to be followed. If ‘NULL‘ (default), all edges have equal weight.

`placement_col` Optional name of a column in ‘edge_list_df‘ holding the page region each link sits in, using the crawler-neutral vocabulary ‘"content"‘, ‘"nav"‘, ‘"header"‘, ‘"footer"‘, ‘"aside"‘. Matching is case-insensitive and whitespace is trimmed. ‘NULL‘ (default) means no placement handling. Placement is **not** a Screaming Frog concept: a per-crawler adapter maps vendor labels onto this vocabulary (see [sf_normalize_position()]) and ‘pagerank()‘ only consumes the result, so any crawler that reports link regions can drive placement-aware scoring.

`accepted_placements`
Optional character vector of placements to retain; edges placed elsewhere (or with a missing placement) are dropped. ‘NULL‘ (default) keeps every edge. Requires ‘placement_col‘.

`placement_weights`
Optional named positive numeric vector assigning edge weights by placement, e.g. ‘c(content = 1, nav = 0.1, header = 0.1, footer = 0.1, aside = 0.1)‘. Placements not named keep weight ‘1‘, so name all five to state a complete recipe. Requires ‘placement_col‘ and cannot be combined with ‘weight_col‘, which it supersedes by building a weight column of its own. Downweighting rather than filtering is deliberate: dropping a region changes the graph’s *shape* (pages reachable only through nav become teleport-only, pages linking out only through nav become dangling), whereas a small weight leaves the topology intact and merely stops the region dominating.

`container_col` Optional name of a column in ‘edge_list_df‘ identifying the **source-side component** each link sits in – the template element the link belongs to, stable across the pages that element appears on. Supplying it switches on the boilerplate detector; ‘NULL‘ (default) leaves it off. Like ‘placement_col‘ this is crawler-neutral data: a per-crawler adapter derives component identity from whatever the crawler reports (a DOM path, a CSS selector, a template ID)

and ‘pagerank()‘ only consumes the result, so any crawler that can identify a link’s component can drive the detector. Cannot be combined with ‘weight_col‘, which it supersedes by building a weight column of its own.

`boilerplate_threshold`

The container-conditioned recurrence ratio at or above which an edge is **classified** boilerplate, in ‘(0, 1]‘. The ratio is the share of pages carrying the container on which that container points at this same target, so ‘1‘ means "every time this component appeared, it linked here" and values near ‘0‘ mean the component chooses a different target on each page. Default ‘0.5‘. Only consulted when ‘container_col‘ is supplied.

`min_container_pages`

Minimum number of pages a container must appear on before any of its edges may be classified. Default ‘10‘. Small containers are excluded because their ratios are quantized – a container on three pages can only score ‘0.33‘, ‘0.67‘ or ‘1‘ – so a high ratio there is thin evidence rather than a strong signal. A judgment call, not a measured cut.

`boilerplate_weight`

The multiplier applied to an edge **classified** boilerplate, in ‘(0, 1]‘. Default ‘0.5‘. Note this is a different quantity from ‘boilerplate_threshold‘ despite sharing a default value: the threshold is a fraction of pages that decides *whether* an edge is boilerplate, this is the discount applied *once it is*. Placement and recurrence are two **detectors feeding one graded axis**, not two independent axes: a nav link is boilerplate by construction, so the factors are not multiplied – that would discount the same link twice for the same fact. The strongest applicable discount wins, giving chrome ‘0.1‘, repetitive in-content ‘0.5‘, and unique in-content ‘1‘. Both factors are recorded separately in the transition audit.

`position_col`

Optional name of a numeric column in ‘edge_list_df‘ holding each link’s **position index** within its source page – ‘1‘ for the first link, ‘2‘ for the second, and so on in reading order. Supplying it switches on the positional-decay axis; ‘NULL‘ (default) leaves it off. This is the genuinely orthogonal axis of the edge-weighting model: where placement and recurrence describe *templatedness* (and feed one graded axis combined by minimum), position describes *reading order* and so composes by **multiplication** – an above-the-fold boilerplate CTA (‘0.5 * 1.0‘) outranks a trailing organic link (‘1.0 * 0.2‘) with no special-casing. Like ‘placement_col‘ and ‘container_col‘ this is crawler-neutral data: the index must be materialized from document order **at ingest**, while it is still trustworthy, and never inferred from row order here, where a filter, join or dedup may already have destroyed it (for Screaming Frog it is read from an **All Outlinks** export, whose row order is document order, never All Inlinks, whose row order is destination-alphabetical). Edges with no index (‘NA‘) keep position weight ‘1‘, so ranking only the source’s main-content links – leaving site chrome to the placement axis – is expressed by indexing only those links. Cannot be combined with ‘weight_col‘, which it supersedes by building a weight column of its own.

`position_transform`

The reading-order decay applied to ‘position_col‘, one of ‘"zipf"‘ (default) or ‘"rank_linear"‘, reusing [transform_weights()] within each source page’s choice set. ‘"zipf"‘ gives ‘weight = 1 / rank^position_alpha‘ (position 1 keeps weight

‘1‘, later positions drop off as a power law); ‘"rank_linear"‘ gives ‘weight = (n - rank + 1) / n‘ across a source’s ‘n‘ indexed links. Only consulted when ‘position_col‘ is supplied.

`position_alpha` The exponent for ‘position_transform = "zipf"‘, a single positive number. Default ‘1‘. Higher values make the drop-off steeper, so position 1 dominates its page more. Unused by ‘"rank_linear"‘.

`position_floor` The smallest position weight, in ‘(0, 1]‘. Default ‘0.01‘. Decayed weights are clamped up to this floor so that compounding the two axes can never reach ‘0‘ – an "effectively dropped" edge must not sneak back in through decay (the same downweight-not-drop rule that governs placement and boilerplate). Only consulted when ‘position_col‘ is supplied.

`duplicate_edge_policy`

How repeated ‘from -> to‘ rows are represented after URL cleaning, redirect/canonical folding, and domain filtering. One of:

‘"collapse"‘ (default) Destination-level surfer: repeated rows collapse to one unweighted destination edge, preserving legacy ‘get_unique_edges()‘ behavior and the common binary PageRank convention.

‘"aggregate"‘ Collapse each ‘from -> to‘ pair with [aggregate_edges()] semantics. Numeric columns, including ‘weight_col‘, are summed; logical columns such as ‘nofollow‘ use the default ‘"any"‘ conflict policy.

‘"count_instances"‘ Link-slot / edge-level surfer: repeated rows increase transition probability. With no ‘weight_col‘, each surviving ‘from -> to‘ pair receives an internal weight equal to its duplicate-row count. With ‘weight_col‘, weights are summed and an ‘instance_count‘ audit column is retained.

`nofollow_col` Optional name of a logical or 0/1 column in ‘edge_list_df‘ indicating nofollow edges. If ‘NULL‘ (default), no nofollow handling is performed.

`nofollow_action`

How to handle nofollow edges when ‘nofollow_col‘ is provided. One of:

‘"evaporate"‘ (default) Nofollow links remain outgoing slots: they consume their weighted share of the source node’s outgoing PR budget but pass nothing to their targets. Implemented via a sink node that absorbs the unpropagated PR.

‘"drop"‘ Remove nofollow edges before allocating the outgoing budget. They consume no slots, so followed edges divide the full budget among themselves.

‘"keep"‘ Retain and follow these edges normally, so their targets receive their allocated shares.

`indexability_df`

Optional data frame mapping URLs to their indexability status (e.g., from an SEO crawl export). See Details.

`indexability_url_col`

Name of the URL column in ‘indexability_df‘. Default ‘"url"‘.

`indexability_status_col`

Name of the status column in ‘indexability_df‘. Default ‘"indexability_status"‘. Values are comma-separated strings; recognized statuses are ‘"Blocked by robots.txt"‘ and ‘"noindex"‘ (case-insensitive for noindex).

`status_df` Optional data frame mapping URLs to their HTTP response status code (e.g., from an SEO crawl export). Lets ‘pagerank()‘ recognize response-dead pages, which would otherwise be scored as ordinary live vertices. See the "HTTP response status" section in Details.

`status_url_col` Name of the URL column in ‘status_df‘. Default ‘"url"‘.

`status_col` Name of the HTTP status-code column in ‘status_df‘. Default ‘"status_code"‘. Values are HTTP status codes (integer, or coercible to integer); codes in ‘400:599‘ mark a page response-dead.

`robots_blocked_action`
How to present robots.txt-blocked pages in results. Both values route the page's throughput to the shared waste sink (no self-loop); they differ only in whether the page itself is shown. One of:

- **‘"show"‘** (default) Blocked pages appear in results showing the authority they collect, useful for seeing wasted PageRank. What they would pass on evaporates to the sink.
- **‘"vanish"‘** Blocked pages are removed from results; their own stationary mass is booked as hidden (their throughput still evaporates to the sink).

`edge_from_col`, `edge_to_col`
Names of from/to columns in ‘edge_list_df‘.

`redirect_from_col`, `redirect_to_col`
Names of from/to columns in ‘redirects_df‘.

`duplicate_from_policy`
How to handle conflicting redirects in ‘redirects_df‘. Passed through to [resolve_redirects()]. Default ‘"strict"‘ (error on conflicts). See [resolve_redirects()] for all available policies.

`loop_handling` How to handle redirect cycles. Passed through to [resolve_redirects()]. Default ‘"error"‘. See [resolve_redirects()] for all available policies.

`canonicals_df` An optional data frame of declared ‘rel=canonical‘ links, with ‘from‘/‘to‘ columns (or as set by ‘canonical_from_col‘ / ‘canonical_to_col‘) pairing a source URL with the canonical it declares. Default ‘NULL‘ (opt-in; the default preserves current behavior). Canonicals are a **distinct, advisory** signal from enforced 3xx ‘redirects_df‘: they are tracked separately and audited via [audit_canonicals()] / [audit_fold()], then folded into the same composed map as redirects (see [build_fold_map()]). Self-canonicals drop as no-ops.

`canonical_from_col`, `canonical_to_col`
From/to columns in ‘canonicals_df‘. Default ‘"from"‘ / ‘"to"‘.

`clean_canonical_urls`
Logical, whether to clean URLs in ‘canonicals_df‘ using the same resolved ‘rurl_params‘ profile as edge and redirect cleaning. Default ‘TRUE‘. Only effective when ‘canonicals_df‘ is provided.

`canonical_duplicate_from_policy`
How to handle a canonical source that declares multiple distinct canonicals. Reuses the ‘duplicate_from_policy‘ enum (see [resolve_redirects()]). Default ‘"strict"‘.

`canonical_loop_handling`
How to handle cycles among declared canonicals. Reuses the ‘loop_handling‘ enum. Default ‘"error"‘.

`canonical_conflict_policy`

How to resolve a redirect-vs-canonical disagreement on the **same source** URL. One of ‘"redirect_wins"‘ (default; the 3xx wins and the canonical on a redirecting source is ignored and flagged), ‘"error"‘ (error on genuine disagreement), or ‘"canonical_wins"‘ (the declared canonical wins for that source, still flagged). See [build_fold_map()].

`out_of_scope_fold`

Policy for composed fold-map entries whose **target** (the representative a source folds onto) is not itself a crawled node. The crawled node set is the unique, non-‘NA‘ edge endpoints captured immediately before folding (indexability URLs are not part of scope). Such an out-of-scope fold silently relabels a crawled page onto an uncrawled URL, inventing a phantom vertex (e.g. a staging crawl whose canonicals all point at the uncrawled production domain). One of:

‘"relabel"‘ (Default) Apply the full fold map unchanged, relabeling crawled sources onto their out-of-scope targets. Preserves historical behavior.

‘"keep"‘ Drop the out-of-scope entries from the fold map before applying it, so crawled source nodes retain their as-crawled identity. The same filtered map is applied to the TIPR prior fold, keeping edges and prior in one namespace.

‘"leak"‘ Treat each such crawled source like an external redirect: route it onto a synthetic **leak sink** node (distinct from the nofollow sink) so the equity flowing INTO it leaves the measured graph ("these pages won't rank; equity goes elsewhere"). The source's own teleport prior is routed to the sink too. The evaporated equity is reported as **leaked mass** in the ‘mass‘ section of the ‘transition_audit‘ (‘reported + sink + hidden + leaked = 1‘). The leak sink is created whenever there is at least one out-of-scope fold, regardless of ‘nofollow_action‘.

Regardless of policy, the count and list of out-of-scope folds (source, target, signal) are recorded in the ‘fold‘ section of the ‘transition_audit‘ object. See [transition_audit].

`keep_domains` Optional character vector of domains to keep. When provided, edges are filtered via [filter_links_by_domain()] so that only links where both endpoints belong to one of the specified domains are included. Useful for restricting to internal links. Default ‘NULL‘ (no domain filtering).

Ordering: filtering runs *after* redirect/canonical folding, so it scopes the post-fold (canonical) namespace, not the crawled input. If an out-of-scope canonical/redirect rewrites the crawled domain onto a different one, filtering on the crawled domain matches nothing (an empty graph). To scope the INPUT you crawled, run [filter_links_by_domain()] on the edge list *before* calling ‘pagerank()‘.

`exclude_domains`

Optional character vector of domains to exclude. Edges where either endpoint belongs to one of these domains are removed. Like ‘keep_domains‘, this filters the post-fold namespace (see the ordering note above). Default ‘NULL‘ (no exclusion).

`keep_hosts` Optional character vector of exact hosts to keep (e.g. ‘"www.example.com"‘), as opposed to registrable domains. Matched on the exact host using the same

canonicalization profile as cleaning, so IDN folding (‘host_encoding‘ in ‘rurl_params‘) applies consistently. Default ‘NULL‘.

`exclude_hosts` Optional character vector of exact hosts to exclude. Edges where either endpoint matches one of these hosts are removed. Ignore rules override keep rules. Default ‘NULL‘.

`prior_df` Optional per-URL external-authority prior for TIPR (authority-weighted teleport). A data frame with one row per URL and a numeric weight. The prior URLs are canonicalized with the same ‘rurl_params‘ and folded through the same redirect map as the edges, weights for URLs that coalesce are summed, and the result is aligned to the final vertex set via [align_prior_to_vertices()] and passed to ‘igraph::page_rank(personalized =)‘. Default ‘NULL‘ (uniform teleport).

The weight column must be an **additive raw count** — the redirect fold sums it (see ‘prior_weight_col‘), which is only meaningful for quantities that add when URLs coalesce. This keeps the prior **source-agnostic**: the default is Ahrefs **referring domains**, but any backlink-source count is a drop-in swap (Ahrefs *links-to-target* or *dofollow-only referring domains*; SEMrush backlink/referring-domain counts; or even non-backlink counts such as GA4 entrances), simply by pointing ‘prior_weight_col‘ at it. Do **not** pass a calculated authority *score* (Ahrefs UR / DR, or any 0–100 rating): scores are not additive (folding two redirect variants is a ‘max‘, not a ‘sum‘), and a per-URL score like UR is itself a PageRank-style metric — using it as a teleport prior for PageRank is circular. See [align_prior_to_vertices()] for the full contract.

`prior_url_col, prior_weight_col`

Column names in ‘prior_df‘. Defaults ‘"url"‘ / ‘"weight"‘. Swapping ‘prior_weight_col‘ between additive count columns is the supported way to A/B alternative authority metrics (e.g. via [pagerank_grid()]); see ‘prior_df‘ for which metrics qualify.

`prior_transform`

How to shape raw authority before it becomes teleport mass. One of ‘"none"‘ (default, faithful linear share), ‘"log"‘, ‘"percentile"‘, ‘"minmax"‘, ‘"zipf"‘, ‘"rank_linear"‘. See [transform_weights()]. Counts are summed on the raw scale before any transform.

`prior_alpha` Mixture weight in ‘[0, 1]‘ between uniform and authority-weighted teleport (‘p = alpha * uniform + (1 - alpha) * authority_share‘). ‘0‘ (default) is pure authority teleport; ‘1‘ reproduces uniform PageRank. See [align_prior_to_vertices()].

`prior_inject_unmatched`

Logical. If ‘TRUE‘, authoritative prior URLs that do not fold onto any existing vertex are added as edge-less isolate vertices so they appear in results carrying their teleport prior. Default ‘FALSE‘ (align-only: such URLs are dropped and logged).

`prior_exclude_waste`

Logical. If ‘TRUE‘ (default), the collect-but-cannot-pass class — noindex, robots-blocked, and 4xx/5xx pages (see ‘indexability_df‘ / ‘status_df‘) — is excluded from the teleport vector: those pages keep the authority that reaches them through inlinks but are no longer paid the uniform teleport share for merely existing. This stops a page from manufacturing authority by linking to many dead

ends (Page & Brin 1998 criticize uniform teleport for "valuing pages simply because they exist"). Set ‘FALSE‘ to give every page uniform teleport, matching ‘igraph::page_rank()‘ for canonical comparisons. Has no effect unless ‘indexability_df‘ or ‘status_df‘ supplies the class; the synthetic evaporation and leak sinks are excluded from teleport regardless.

`prior_verbose` Logical, whether to emit prior-alignment coverage diagnostics. Default ‘TRUE‘. Only relevant when ‘prior_df‘ is supplied.

`damping` The PageRank damping factor α (the random surfer's continue probability; the teleport probability is $1 - \alpha$). A single number in ‘[0, 1]‘, default ‘0.85‘ — the field convention from Brin & Page. Forwarded to [compute_pagerank()] and on to ‘igraph::page_rank()‘. Higher values weight the link structure more heavily but converge more slowly: a power-iteration solve needs roughly $\log_{10}(\tau)/\log_{10}(\alpha)$ iterations to reach residual τ, so pushing α toward 1 sharply raises the iteration count (raise ‘niter‘ accordingly when using the ARPACK solver). See the "Damping factor" section in Details for guidance on choosing it, and [damping_sensitivity()] to sweep a range of values.

`...` Additional arguments passed to [compute_pagerank()] and subsequently to ‘igraph::page_rank()‘. Besides ‘damping‘, the recognized convergence controls ‘algo‘ (‘"prpack"‘ / ‘"arpack"‘), ‘eps‘, and ‘niter‘ are forwarded here; see the "Convergence controls" section below.

`preset` Optional named argument bundle describing a common view of the graph: a preset name (‘"raw"‘, ‘"declared"‘, ‘"reversed"‘, ‘"content"‘), a [pr_preset()] result, or ‘NULL‘ (default, no preset). Preset values are applied only to arguments you did not name yourself, so precedence is **explicit argument > preset > base default**. Must be named in full (it sits after ‘...‘). See [pr_preset()] for the exact expansion of each preset.

Details

Damping factor

The ‘damping‘ factor α is the probability that the random surfer follows a link rather than teleporting; the remaining $1 - \alpha$ is spread over the teleport vector (uniform, or the supplied TIPR ‘prior_df‘).

The default ‘0.85‘ is the original Brin & Page value and remains the field convention, but it is *eminently empirical* — Boldi, Santini & Vigna (*PageRank as a Function of the Damping Factor*, WWW 2005) show it has no analytical claim to being uniquely correct. A common misconception is that values close to 1 yield "more accurate" rankings by trusting the link graph more; for real-world graphs they instead make the ranking dominated by the graph's largest near-cyclic component and, in the limit $\alpha \to 1$, degenerate rather than converge to a more meaningful order.

Raising α also degrades convergence sharply. A power-iteration solve needs about $\log_{10}(\tau)/\log_{10}(\alpha)$ iterations to reach residual τ (Langville & Meyer, *Deeper Inside PageRank*, Internet Mathematics 2004). At $\tau = 10^{-8}$: $\alpha = 0.85$ needs ~114 iterations, $\alpha = 0.95$ ~362, and $\alpha = 0.99$ ~1,833 — so a high damping factor is both slower and rarely better. When you do raise it on the ARPACK solver, raise ‘niter‘ to match (see "Convergence controls" below).

Both of those papers study the open web. Whether ‘0.85‘ is still the right convention for a site-scale intranet graph is an open empirical question; [damping_sensitivity()] sweeps a range of α values so you can see how much the ranking on *your* graph actually moves.

Convergence controls

‘igraph::page_rank()‘ is called through one of two solver back-ends, selected with ‘algo‘ (forwarded via ‘...‘):

‘"prpack"‘ (default) A fast, exact direct solver. It has **no** tunable tolerance or iteration cap, and reports no iteration count.

‘"arpack"‘ An iterative eigensolver that honors ‘eps‘ (the L1 tolerance) and ‘niter‘ (the maximum iterations), and reports how many iterations it used.

Modern ‘igraph‘ (2.x) removed the legacy ‘page_rank()‘ ‘eps‘ / ‘niter‘ arguments; this package re-exposes them as friendly aliases for the ARPACK ‘options$tol‘ / ‘options$maxiter‘ controls. Because PRPACK ignores them, supplying either ‘eps‘ or ‘niter‘ transparently switches ‘algo‘ to ‘"arpack"‘. As a rule of thumb a power-iteration solve needs about ‘log10(eps) / log10(damping)‘ iterations, so raise ‘niter‘ when you push ‘damping‘ toward 1.

Every non-empty result carries a ‘"convergence"‘ attribute (a [pagerank_convergence] object) reporting the solver, iteration count (when the solver exposes it), and the solver-independent post-hoc L1 residual $\|Gx - x\|_1$ of the returned vector. Retrieve it with ‘attr(result, "convergence")‘.

The waste class (noindex, robots-blocked, response-dead)

‘pagerankr‘ models one **"collects PageRank but cannot pass it"** class and routes every member through a single shared **waste sink** with the same mechanism: the member loses all of its outgoing edges and gains exactly one edge to the sink, so it still absorbs the authority its inlinks send but passes none of it back into the graph. The sink is an internal accounting bucket (never a page, always stripped from the returned result); the mass it collects is reported as **evaporated** mass in the transition audit. Removing the old robots-blocked self-loop is deliberate: a self-loop is an absorbing rank sink that compounds inbound authority every iteration (a measured 8.3× inflation), whereas the waste sink lets authority flow in and stop.

Members come from three signals:

noindex (‘indexability_df‘): ‘pagerankr‘ models the ranked corpus as the set of indexed documents, so a noindex page is outside it — it may receive authority through inlinks but cannot redistribute it within the indexed graph. This is a PageRank modeling choice; it does not assert that Google defines noindex as a nofollow directive. noindex routing to the sink is independent of ‘nofollow_action‘ (which governs only real ‘rel=nofollow‘ edges): a noindex page always routes to the sink. noindex pages still appear in results so their received authority remains auditable.

robots.txt-blocked (‘indexability_df‘): Google cannot access the page content, so there are no visible outgoing links. ‘robots_blocked_action‘ controls only whether the page appears in results (‘"show"‘, the default) or is removed with its own mass booked as hidden (‘"vanish"‘) — both route the page’s throughput to the sink.

Priority rule: robots.txt always takes precedence over noindex. If a page is both robots-blocked and noindex, it is treated as robots-blocked.

HTTP response status

When ‘status_df‘ is provided, pages whose HTTP status code falls in ‘400:599‘ are recognized as **response-dead**: at crawl time they returned no content and expose no outgoing links, so they can collect authority through their inlinks but cannot pass any of it on. They belong to the same waste class as noindex pages and route to the same sink; because a dead page typically has no outlinks, this ADDS the one edge to the sink that stops it from dangling and recycling its inbound authority to every page via teleport.

‘pagerankr‘ does **not** split 4xx from 5xx. The crawl is a snapshot, and at crawl time a transient ‘503‘ and a permanent ‘404‘ are indistinguishable: both return no content and expose no links. Modeling one as recoverable would require guessing about a future the crawl has no data on — the same reason ‘pagerankr‘ folds a ‘302‘ exactly like a ‘301‘. A caller who knows a given ‘5xx‘ was a blip should re-crawl rather than have the tool assume recovery on its behalf.

‘3xx‘ redirects are **not** part of this class; they are modeled through ‘redirects_df‘. Codes below ‘400‘, and rows whose status is missing or cannot be parsed as an integer, are treated as live. Response-dead pages that are present in the graph are counted in the returned ‘transition_audit‘ (‘config$has_status‘ and ‘n_status_dead‘).

Reverse / inverse PageRank (‘reverse = TRUE‘)

Standard PageRank measures **inflow** importance ("who points to me"). With ‘reverse = TRUE‘ the link graph is transposed before computation, yielding **outflow centrality** ("does this page funnel authority outward"). This is the *reverse PageRank* of Bar-Yossef & Mashiach (CIKM 2008), equivalent to the **CheiRank** of the transposed Google matrix, and the PageRank-flavored analogue of the *hub* score in Kleinberg's HITS. The sibling ‘semantic‘ project consumes this as an outflow signal.

Only edge orientation is flipped; URL cleaning, redirect folding, duplicate-edge policy, edge weights, domain/host filtering, and the teleport prior all behave identically (they are direction-agnostic). To obtain it directly from an edge list, swapping the from/to columns and running ordinary ‘pagerank()‘ is equivalent — ‘reverse = TRUE‘ just performs that flip internally so weight, redirect, and sink handling cannot be mis-wired by a manual swap.

This is unrelated to the TIPR / personalized-prior feature (‘prior_df‘). That seeds the *teleport* vector with external authority (e.g. backlinks) but still computes inflow PageRank on the forward graph; ‘reverse‘ is a pure *graph operation* on edge direction. The two are orthogonal and may be combined.

Direction-sensitive features are rejected under ‘reverse = TRUE‘ because their semantics do not transpose:

‘nofollow_action = "evaporate"‘ Errors. The evaporation sink models a *source* wasting its outgoing budget; reversed, it would inject rank instead. Use ‘"drop"‘ — the correct treatment of a nofollowed link for outflow centrality, since it funnels no authority outward — or ‘"keep"‘.

‘indexability_df‘ Errors. noindex and robots.txt blocking (route the page's outgoing budget to the waste sink) encode forward crawl/index behavior with no meaningful transpose.

Duplicate edge policy

The original PageRank papers define a page's vote as divided by its outgoing link count but do not pin down how repeated hyperlinks from one source page to the same target are represented. The standard textbook / binary operationalization treats the outgoing set as a destination relation, so multiple ‘A -> C‘ rows collapse to one destination edge. ‘pagerankr‘ keeps that as the default (‘duplicate_edge_policy = "collapse"‘) for backward compatibility and as the less spam-sensitive model.

Weighted / multigraph PageRank is also valid when repeated link slots are the intended unit. Use ‘duplicate_edge_policy = "count_instances"‘ for a link-slot surfer: ‘A -> B, A -> C, A -> C‘ sends twice as much outgoing mass to ‘C‘ as to ‘B‘, equivalent to explicit weights ‘B = 1, C = 2‘ and to igraph's treatment of parallel edges. Use ‘"aggregate"‘ when duplicate rows should be collapsed loss-aware, especially with an existing ‘weight_col‘; numeric duplicate weights are summed instead of silently keeping the first row.

Fold-then-filter ordering (domain / host scope)

Redirect and canonical folding runs **before** the ‘keep_domains‘ / ‘exclude_domains‘ / ‘keep_hosts‘ / ‘exclude_hosts‘ filter. Folding can rewrite the node namespace: an out-of-scope canonical (e.g. every ‘staging.example.dev‘ page declaring a ‘example.com‘ canonical) relabels crawled nodes onto a domain you never crawled. Because the filter then sees only the post-fold (canonical) namespace, filtering on the domain you actually crawled matches nothing and returns an empty graph.

‘pagerank()‘ detects this specific case – a filter value that classified one or more crawled (pre-fold) nodes but no surviving post-fold node – and emits an actionable ‘warning()‘ naming the folded-away value(s) and pointing at the out-of-scope fold as the cause. This is a diagnostic only; the fold-then-filter order is unchanged.

To scope the **input you crawled**, filter first: run [filter_links_by_domain()] on the edge list (and, if used, the redirect / canonical data frames) *before* calling ‘pagerank()‘. To scope the folded graph, filter on the post-fold (canonical) domain/host instead.

Value

A data frame with node names and their PageRank scores. When nofollow evaporation, the waste class (noindex / robots-blocked / response-dead), or ‘robots_blocked_action = "vanish"‘ is active, the returned scores may sum to less than 1. The difference is not undifferentiated "leakage": it is decomposed into **evaporated mass** (authority sent to the shared waste sink, i.e. what the class and every real nofollowed link passed on but could not deliver), **leaked mass** (authority sent to the leak sink under ‘out_of_scope_fold = "leak"‘), and **hidden mass** (the own stationary mass of robots-blocked nodes removed from the results). The full breakdown — reported / evaporated (sink) / leaked / hidden / total (= 1) — is recorded in the ‘mass‘ field of the transition audit (see below).

When ‘indexability_df‘ or ‘status_df‘ is supplied, the result gains two per-URL waste-attribution columns, present only with those inputs (mirroring how ‘prior_weight‘ appears only with ‘prior_df‘), so the result is otherwise unchanged:

‘page_state‘ The page’s health/indexability state: ‘"live"‘, ‘"noindex"‘, ‘"robots_blocked"‘, or ‘"response_dead"‘ (robots-blocked > response-dead > noindex > live when a page carries more than one signal).

‘wasted_mass‘ The authority the page collected and black-holed — its share of the shared waste sink’s stationary mass. A waste-class page routes its whole throughput to the absorbing sink, so this is ‘damping / (1 - damping)‘ times its own reported score: larger than, and distinct from, that score, which answers the "how much did this page amass and evaporate" question ‘page_state‘ only labels. It sums across the waste class to the evaporated mass reported in the transition audit (‘mass$sink‘). A ‘"live"‘ page routes nothing to the sink, so its ‘wasted_mass‘ is ‘0‘.

‘page_state‘ is the page’s *health* state; the ‘node_status‘ column returned by [simulate_changes()] is a distinct axis — a node’s *role in a before/after comparison* (‘normal‘ / ‘new-target‘ / ‘removed-dead‘) — not a second name for the same thing.

The data frame additionally carries a ‘"transition_audit"‘ attribute (a [transition_audit] object) recording how the transition graph was built: row/edge counts, behavioral-weight coverage, normalization totals, the page-mass decomposition (reported / evaporated / leaked / hidden / total), dropped data (rows lost to NA / dedup / self-loops, unmatched prior URLs), and the model configuration used. Retrieve it with ‘attr(result, "transition_audit")‘.

The result additionally carries a ‘"convergence"‘ attribute (a [pagerank_convergence] object); see the "Convergence controls" section.

Examples

```
# Basic example
edges <- data.frame(
  from = c("http://A.com/", "B", "C?q=1", "D"),
  to = c("B", "http://A.com", "D#frag", "D")
)
redirects <- data.frame(
  from = c("C?q=1", "B"),
  to = c("http://C_resolved.com", "A") # B redirects to A, C to C_resolved
)

# Run full pipeline
pr_full <- pagerank(
  edges,
  redirects_df = redirects, self_loops = "drop", drop_isolates_flag = TRUE
)
print(pr_full)

# Run without URL cleaning for edges
# (warning expected if query params present)
pr_no_edge_clean <- pagerank(
  edges,
  redirects_df = redirects, clean_edge_urls = FALSE
)
print(pr_no_edge_clean)

# Keep isolates
edges_isol <- rbind(edges, data.frame(from = "ISO", to = "LAND"))
pr_keep_isolates <- pagerank(edges_isol, drop_isolates_flag = FALSE)
print(pr_keep_isolates)

# With nofollow edges (evaporate mode)
edges_nf <- data.frame(
  from = c("A", "A", "B"), to = c("B", "C", "A"),
  nofollow = c(FALSE, TRUE, FALSE)
)
pr_nf <- pagerank(edges_nf,
  nofollow_col = "nofollow",
  nofollow_action = "evaporate", clean_edge_urls = FALSE
)
print(pr_nf)

# Reverse / inverse PageRank (outflow centrality, a.k.a. CheiRank):
# a page that funnels authority outward scores high.
pr_reverse <- pagerank(edges, redirects_df = redirects, reverse = TRUE)
print(pr_reverse)
```

`pagerank_convergence` *PageRank convergence diagnostic object*

Description

Records how the PageRank stationary vector was obtained: which solver ran, how many iterations it used (when that is observable), and how well the returned vector satisfies the PageRank fixed-point equation. It is attached to the result of [compute_pagerank()] / [pagerank()] as the ‘"convergence"‘ attribute and is the companion to the [transition_audit][transition_audit] provenance record.

Details

Why the solver matters

‘igraph::page_rank()‘ offers two back-ends:

‘"prpack"‘ (default) A direct/sparse solver (the PRPACK library). It is fast and exact to machine precision, but it is **not** iterative in any way it exposes: there is no iteration count and no tolerance knob, so ‘iters‘ is reported as ‘NA‘ and ‘eps‘ / ‘niter‘ have no effect.

‘"arpack"‘ An iterative eigensolver. It honors a tolerance and a maximum iteration count and reports the iterations it actually used and whether it converged. This is the only back-end on which ‘eps‘ and ‘niter‘ take effect; supplying either to [compute_pagerank()] / [pagerank()] therefore transparently selects it.

The old ‘igraph‘ ‘eps‘ / ‘niter‘ arguments to ‘page_rank()‘ were removed in modern ‘igraph‘ (2.x); this package re-introduces them as friendly aliases that map onto the ARPACK ‘options$tol‘ / ‘options$maxiter‘ controls.

The residual is solver-independent

Regardless of back-end, ‘residual‘ is computed here directly from the returned vector as the L1 norm of one PageRank operator application, $\|Gx - x\|_1$, where G is the Google operator implied by the scored graph, the damping factor, and the teleport vector (uniform, or the supplied TIPR prior). This is the standard Kamvar, Haveliwala & Golub (2004) stopping criterion, evaluated *post hoc* so it is a genuine, comparable quality check across both solvers (a converged solution sits near machine precision). ‘tol_met‘ reports whether ‘residual‘ is at or below ‘tol‘ (the supplied ‘eps‘, or the conventional default of ‘1e-3‘ when ‘eps‘ is ‘NULL‘), additionally requiring ‘info == 0‘ for the ARPACK back-end.

Iteration-count rule of thumb

Power-iteration PageRank needs about $\log_{10}(\tau)/\log_{10}(\alpha)$ iterations to reach residual τ at damping α (Langville & Meyer, 2004). At $\tau = 10^{-8}$: $\alpha = 0.85$ needs ~114, $\alpha = 0.95$ ~362, and $\alpha = 0.99$ ~1,833 iterations — so a high damping factor degrades convergence sharply. ARPACK is not plain power iteration, so its reported ‘iters‘ is typically far lower, but the same qualitative warning applies: raise ‘niter‘ if you raise the damping factor toward 1.

See Also

[compute_pagerank()], [pagerank()], [transition_audit]

Examples

```
edges <- data.frame(
  from = c("A", "B", "C", "A"),
  to = c("B", "C", "A", "C")
)

# The default PRPACK back-end is direct, so it reports no iteration count.
# The residual is still computed post hoc, so it stays comparable.
conv <- attr(compute_pagerank(edges), "convergence")
conv
conv$algo
conv$iters # NA: PRPACK does not expose iterations
conv$tol_met

# Supplying eps / niter transparently selects the iterative ARPACK
# back-end (with a message), which honors the tolerance and reports the
# iterations it used. Pass algo = "arpack" explicitly to silence it.
conv_arpack <- attr(
  compute_pagerank(edges, eps = 1e-10, niter = 1000), "convergence"
)
conv_arpack$algo
conv_arpack$iters
conv_arpack$tol
```

---

pagerank_grid *Run PageRank Across Multiple Parameter Sets*

Description

Executes [pagerank()] for each entry in a named parameter grid, returning a single combined data frame with a ‘model_id‘ column identifying which configuration produced each row.

Usage

```
pagerank_grid(
  edge_list_df,
  params_grid,
  redirects_df = NULL,
  ...,
  edge_from_col = "from",
  edge_to_col = "to"
)
```

Arguments

edge_list_df	A data frame representing the edge list (passed to every [pagerank()] call).
params_grid	A named list of named lists. Each inner list contains parameter overrides for [pagerank()]. The top-level names become the ‘model_id‘ values in the output.

redirects_df An optional redirect data frame (passed to every call). Default ‘NULL‘.

... Common parameters shared across all models (e.g., ‘clean_edge_urls‘, ‘rurl_params‘, ‘damping‘). These are passed to every [pagerank()] call and can be overridden by entries in ‘params_grid‘.

edge_from_col, edge_to_col
Names of from/to columns in ‘edge_list_df‘. Default ‘"from"‘ / ‘"to"‘.

Value

A data frame with columns ‘model_id‘, ‘node_name‘, and ‘pagerank‘ (or the column names returned by [pagerank()]). Rows from different models are stacked via [rbind()].

Examples

```
edges <- data.frame(
  from = c("A", "B", "C", "A"),
  to = c("B", "C", "A", "C")
)
params <- list(
  baseline = list(damping = 0.85, self_loops = "drop"),
  high_damp = list(damping = 0.95, self_loops = "drop"),
  keep_loops = list(damping = 0.85, self_loops = "keep")
)
grid <- pagerank_grid(edges, params, clean_edge_urls = FALSE)
print(grid)

# Compare two models from the grid
baseline <- grid[grid$model_id == "baseline", ]
high_damp <- grid[grid$model_id == "high_damp", ]
compare_pagerank(baseline, high_damp)
```

---

pagerank_metrics *PageRank Distribution Metrics*

Description

Compute summary statistics for a vector of PageRank scores. These metrics help characterize how concentrated or dispersed the PageRank distribution is, which is useful when comparing different models or parameter configurations.

Usage

```
pr_gini(x)

pr_entropy(x)

pr_top_k_share(x, k = 0.1)
```

Arguments

x	Numeric vector of non-negative values (typically PageRank scores).
k	Fraction of nodes to consider (0 < k <= 1). Default ‘0.1‘ (top 10 percent).

Value

A single numeric value.

A single numeric value (in nats). Returns ‘NA‘ for empty or all-zero inputs.

A single numeric value between 0 and 1 representing the cumulative share.

Functions

- `pr_gini()`: Gini coefficient (0 = perfectly equal, 1 = maximally concentrated).
- `pr_entropy()`: Shannon entropy (higher = more uniform distribution).
- `pr_top_k_share()`: Share of total PageRank held by the top-k fraction of nodes (e.g., top 10 percent).

Examples

```
pr_gini(c(0.5, 0.3, 0.2))
pr_gini(c(1, 0, 0))
pr_entropy(c(1 / 3, 1 / 3, 1 / 3)) # maximum entropy for 3 nodes
pr_entropy(c(1, 0, 0)) # minimum entropy
pr_top_k_share(c(0.5, 0.3, 0.1, 0.05, 0.05))
pr_top_k_share(c(0.5, 0.3, 0.1, 0.05, 0.05), k = 0.4)
```

---

pagerank_screaming_frog

Score a Screaming Frog bundle with PageRank

Description

Thin convenience wrapper around [pagerank()] for the stable [screaming_frog_bundle()] handoff object. Only ‘bundle$edges‘ enter the graph. Raw observations and dedicated redirect, canonical, indexability, and resource rows stay out of the edge list and are mapped to the existing [pagerank()] arguments.

Usage

```
pagerank_screaming_frog(
  bundle,
  accepted_placements = NULL,
  link_origins = NULL,
  placement_weights = NULL,
  weight_col = NULL,
  apply_canonicals = TRUE,
```

```
  apply_redirects = TRUE,
  preset = NULL,
  ...
)
```

Arguments

bundle	A ‘screaming_frog_bundle‘ object.
accepted_placements, placement_weights	Placement controls forwarded to [pagerank()], which owns them: placement is a crawler-neutral concept, and this wrapper only supplies the bundle’s normalized ‘placement‘ column as [pagerank()]’s ‘placement_col‘. See [pagerank()] for the vocabulary (‘"content"‘, ‘"nav"‘, ‘"header"‘, ‘"footer"‘, ‘"aside"‘) and semantics.
link_origins	Optional character vector of normalized link origins to retain. Values must be among ‘"html"‘, ‘"rendered"‘, and ‘"html_rendered"‘. ‘NULL‘ keeps all origins present in ‘bundle$edges‘. Unlike placement, link origin *is* a Screaming Frog concept and stays wrapper-owned.
weight_col	Optional existing edge weight column, forwarded to [pagerank()]. Cannot be combined with ‘placement_weights‘.
apply_canonicals	Logical flag (default ‘TRUE‘). When ‘TRUE‘ the bundle’s ‘rel=canonical‘ signals are folded into the graph via [pagerank()]’s ‘canonicals_df‘. Set ‘FALSE‘ for an as-crawled run that preserves the crawled node identities (no canonical folding) — useful when canonicals point off the crawled domain (e.g. a mirror/staging host) and would otherwise relabel crawled pages onto uncrawled targets.
apply_redirects	Logical flag (default ‘TRUE‘). When ‘TRUE‘ the bundle’s redirect signals are folded into the graph via [pagerank()]’s ‘redirects_df‘. Set ‘FALSE‘ to skip redirect folding and keep the as-crawled node identities.
preset	Optional named view forwarded to [pagerank()]’s ‘preset‘, one of ‘"raw"‘, ‘"declared"‘, ‘"reversed"‘, ‘"content"‘, or a [pr_preset()] result. The ‘"raw"‘ view additionally switches off the bundle’s declared canonical, redirect and indexability tables, since "the graph exactly as crawled" cannot honor declarations the wrapper would otherwise feed in. An explicit ‘apply_canonicals‘ or ‘apply_redirects‘ still overrides the ‘"raw"‘ default for that table. See ‘vignette("presets")‘.
...	Additional scoring controls passed to [pagerank()], such as ‘self_loops‘, ‘drop_isolates_flag‘, ‘nofollow_action‘, ‘robots_blocked_action‘, ‘rurl_params‘, prior settings, and ‘damping‘. Positional decay is opt-in here too: the bundle’s ‘edges‘ carry a ‘position_index‘ column (each link’s reading-order rank among its source page’s content links, materialized only from an **All Outlinks** export), so pass ‘position_col = "position_index"‘ – optionally with ‘position_transform‘ / ‘position_alpha‘ / ‘position_floor‘ – to switch the axis on. It stays off by default, per the faithful-default rule, because reading-order decay reshuffles ranking as hard as placement weighting does.

Value

The [pagerank()] result data frame. It retains the ‘"transition_audit"‘ attribute from [pagerank()] and adds a ‘"screaming_frog_import"‘ attribute containing bundle diagnostics, provenance, and wrapper filtering/weighting choices.

Examples

```
internal <- data.frame(
  Address = c("https://example.com/", "https://example.com/a"),
  `Status Code` = c("200", "200"),
  check.names = FALSE
)
links <- data.frame(
  Type = "Hyperlink",
  Source = "https://example.com/",
  Destination = "https://example.com/a",
  Follow = "TRUE",
  check.names = FALSE
)
bundle <- screaming_frog_bundle(internal, links, "all_outlinks")
pagerank_screaming_frog(bundle)
```

---

pagerank_stability *Report how stable a PageRank ranking is across damping factors*

Description

Sweeps [pagerank()] over a grid of damping factors with [damping_sensitivity()] and compares each α's ranking against a reference α with [compare_pagerank()], returning a one-row-per- α stability summary. It answers the open question flagged in the "Damping factor" section of [pagerank()]: on *your* graph, how much does the ranking actually move as α varies?

Usage

```
pagerank_stability(
  edge_list_df,
  alphas = c(0.75, 0.8, 0.85, 0.9, 0.95),
  reference = 0.85,
  top_k = 10,
  ...
)
```

Arguments

edge_list_df	A data frame representing the edge list, passed to every [pagerank()] call. (Named for consistency with the rest of the package; it is an edge list, not a constructed graph object.)

alphas	Numeric vector of damping factors to sweep, each strictly between 0 and 1. Default 'c(0.75, 0.80, 0.85, 0.90, 0.95)'. Duplicate values are dropped.
reference	The baseline damping factor every other α is compared against. A single number strictly between 0 and 1, default '0.85'. Included in the sweep automatically if not already in 'alphas'.
top_k	Size of the top-scoring set used for the 'top_k_overlap' churn metric. Positive integer, default '10'.
...	Additional arguments forwarded to [damping_sensitivity()] and on to [pagerank()] (e.g. 'redirects_df', 'weight_col', 'algo', 'prior_df'). Passing 'damping' is an error, since 'alphas' drives the damping factor.

Details

A Spearman rank correlation near 1 across the whole grid means the choice of damping factor is immaterial for this graph — the conventional '0.85' is as good as any nearby value. A low correlation, or a top-k overlap well below 1, flags a graph whose ranking is genuinely α-sensitive and worth investigating before trusting any single solve.

This is a thin orchestration layer: it performs no PageRank math of its own, delegating the solves to [damping_sensitivity()] and the rank-comparison statistics to [compare_pagerank()]. The 'reference' factor is always included in the sweep (even if absent from 'alphas') so it can serve as the comparison baseline; its own row is a sanity anchor ('spearman_rho = 1', 'mean_abs_delta = 0', 'top_k_overlap = 1').

Value

A data frame with one row per swept α (ascending), with columns:

'alpha' The damping factor.

'spearman_rho' Spearman rank correlation of this α's ranking against the reference, on their common nodes ('NA' if fewer than 3 common nodes).

'mean_abs_delta' Mean absolute score difference vs the reference on common nodes.

'top_k_overlap' Fraction in '[0, 1]' of the reference's top-'k' pages that are also in this α's top-'k' (1 = identical top set). The effective 'k' shrinks to the node count on small graphs.

'nodes_gained', 'nodes_lost' Nodes present at this α but not the reference, and vice versa. Normally 0: varying α changes scores, not the node set.

'algo', 'iters', 'iters_estimate', 'residual', 'tol', 'converged', 'n_nodes' The per-α convergence metadata carried over from [damping_sensitivity()].

The full per-(URL, α) sensitivity frame from [damping_sensitivity()] is attached as the '"sensitivity"' attribute, and the 'reference' and 'top_k' used are attached as same-named attributes.

See Also

[damping_sensitivity()], [compare_pagerank()], [pagerank()] (the "Damping factor" section)

Examples

```
edges <- data.frame(
  from = c("A", "B", "C", "A", "D"),
  to = c("B", "C", "A", "C", "A")
)
stab <- pagerank_stability(edges, clean_edge_urls = FALSE)
print(stab)

# Drill into the per-(url, alpha) scores behind the summary.
head(attr(stab, "sensitivity"))
```

---

print.pagerank_convergence

Print a pagerank_convergence object

Description

Print a pagerank_convergence object

Usage

```
## S3 method for class 'pagerank_convergence'
print(x, ...)
```

Arguments

x	A ‘pagerank_convergence‘ object.
...	Unused; for S3 compatibility.

Value

‘x‘, invisibly.

Examples

```
edges <- data.frame(
  from = c("A", "B", "C"), to = c("B", "C", "A")
)
pr <- compute_pagerank(edges)
conv <- attr(pr, "convergence")
print(conv)
```

print.screaming_frog_bundle *Print a Screaming Frog bundle*

Description

Print a Screaming Frog bundle

Usage

```
## S3 method for class 'screaming_frog_bundle'
print(x, ...)
```

Arguments

x	A ‘screaming_frog_bundle‘ object.
...	Unused; for S3 compatibility.

Value

‘x‘, invisibly.

Examples

```
internal <- data.frame(
  Address = c("https://example.com/", "https://example.com/a"),
  `Status Code` = c("200", "200"),
  check.names = FALSE
)
links <- data.frame(
  Type = "Hyperlink",
  Source = "https://example.com/",
  Destination = "https://example.com/a",
  Follow = "TRUE",
  check.names = FALSE
)
bundle <- screaming_frog_bundle(internal, links, "all_outlinks")
print(bundle)
```

print.transition_audit *Print a transition_audit object*

Description

Print a transition_audit object

Usage

```
## S3 method for class 'transition_audit'
print(x, ...)
```

Arguments

x	A ‘transition_audit‘ object.
...	Unused; for S3 compatibility.

Value

‘x‘, invisibly.

Examples

```
# A transition_audit is attached to every pagerank() result; print it to
# get a human-readable construction / provenance summary.
edges <- data.frame(from = c("a", "a", "b"), to = c("b", "c", "c"))
result <- pagerank(edges)
audit <- attr(result, "transition_audit")
print(audit)
```

pr_preset *Named argument bundles for common PageRank views*

Description

[pagerank()] exposes many graph-preparation knobs. A **preset** is a small, named bundle of those arguments describing one recurring *view* of the link graph, so a view is a one-liner instead of a hand-assembled argument list.

Usage

```
pr_preset(name)
```

Arguments

name	A single string naming a registered preset. See "Presets".

Details

‘pr_preset()‘ returns the bundle as a plain named list, so it is inspectable (print it to audit exactly what a preset does) and spliceable (‘do.call(pagerank, c(list(edges), pr_preset("raw")))‘). Passing the name directly – ‘pagerank(edges, preset = "raw")‘ – is equivalent.

Value

A named list of [pagerank()] arguments.

Presets

‘"raw"‘ The graph exactly as crawled: nothing is applied. Self loops are kept, isolates are kept, ‘rel=nofollow‘ is ignored (the edge votes like any other), and fold-map entries pointing at uncrawled targets are dropped rather than relabeling crawled pages onto phantom vertices. This is the faithful baseline to compare every other view against.

‘"declared"‘ The graph after honoring the signals the site *declares*: ‘rel=nofollow‘ evaporates, declared canonical/redirect targets are followed even when they were not crawled, robots-blocked pages keep the authority they collect, and self loops and isolates are dropped. This is a **pure pin of the package defaults** – it changes nothing about how [pagerank()] behaves today. Its value is that it *states* the default view: a run made with ‘preset = "declared"‘ is a recorded, auditable claim about which view was intended, and it stays pinned to the bundle as documented even if a future default moves. Note that the declared **data** (‘redirects_df‘, ‘canonicals_df‘, ‘indexability_df‘) still has to be supplied by the caller – a preset sets policy, never data.

‘"reversed"‘ The graph with every edge flipped (‘reverse = TRUE‘), yielding reverse / inverse PageRank: a page scores highly when it points *at* well-connected pages rather than when it is pointed at. The feeder view. Note that [topic_feeder_pagerank()] already reverses the graph itself, so this preset is a no-op there rather than an error.

‘"content"‘ Weights edges by the page region they were found in: links in the main content keep their full vote, while links in navigation, header, footer and aside are discounted to a tenth. Site chrome is typically the large majority of a crawl’s edges, so left unweighted it *manufactures* the ranking. Edges are **downweighted, never dropped** – placement is a heuristic classification, so a misclassified content link at 0.1 is a small error where a dropped one is a silent deletion, and dropping most of the graph would also manufacture isolates and dangling pages. All five placement terms are named explicitly, so this is a complete recipe rather than a partial adjustment.

This preset sets policy; the *data* is ‘placement_col‘, which the caller must supply (it errors otherwise). [pagerank_screaming_frog()] supplies it from the bundle, so ‘preset = "content"‘ works there directly.

Presets are not composable with one another – ‘preset‘ takes a single bundle. ‘"content"‘ sets only placement weights and leaves every graph hygiene knob at its default, which *is* the ‘"declared"‘ view, so the two do not need to be combined.

Provenance

The [transition_audit] attached to a [pagerank()] result records which preset produced it, in ‘audit$config$preset‘: the preset name for a registered preset (whether passed by name or as a ‘pr_preset()‘

result), '"custom"' for a hand-rolled bundle, and 'NULL' when no preset was used. The rest of 'config' records the resulting configuration, so a result can be both reconstructed and attributed to the named view it was asked for.

Precedence

Arguments resolve **explicit argument > preset > base default**. A preset value is applied only to arguments the caller did not name, so an explicit argument is never silently overridden:

"'r pagerank(edges, preset = "raw") # nofollow kept pagerank(edges, preset = "raw", nofollow_action = "drop") # "drop" wins "'

This holds through the wrappers that forward '...' to [pagerank()] ([trustrank()], [topic_sensitive_pagerank()], [topic_feeder_pagerank()], [pagerank_screaming_frog()]), with one boundary: arguments a wrapper sets itself are wrapper-owned and a preset cannot change them.

See Also

'vignette("presets")' for each preset's full expansion, worked examples, and the precedence rule.

Examples

```
pr_preset("raw")
pr_preset("declared")

edges <- data.frame(from = c("A", "B"), to = c("B", "C"))

# Equivalent ways to run the raw view
pagerank(edges, preset = "raw")
do.call(pagerank, c(list(edges), pr_preset("raw")))

# An explicit argument always wins over the preset
pagerank(edges, preset = "raw", drop_isolates_flag = TRUE)

# "content" needs a placement column to read regions from. B is linked from
# the main content and C only from the footer, so B outranks C.
placed <- data.frame(
  from = c("A", "A", "B", "C"),
  to = c("B", "C", "A", "A"),
  region = c("content", "footer", "content", "content")
)
pagerank(placed, preset = "content", placement_col = "region")
```

---

resolve_canonicals *Resolve edge endpoints through rel=canonical declarations*

Description

Applies declared 'rel=canonical' folds to the source and target columns of an edge list. This is URL **folding** from canonical-link signals, not URL syntax canonicalization such as lower-casing hosts or removing tracking parameters.

Usage

```
resolve_canonicals(
  edge_list_df,
  canonicals_df,
  edge_from_col = "from",
  edge_to_col = "to",
  canonical_from_col = "from",
  canonical_to_col = "to",
  canonical_duplicate_from_policy = c("strict", "first_wins", "last_wins",
    "most_frequent", "prune_source", "resolve_if_consistent"),
  canonical_loop_handling = c("error", "prune_loop", "break_arrow")
)
```

Arguments

`edge_list_df`	A data frame representing the edge list.
`canonicals_df`	A data frame of declared canonical links.
`edge_from_col, edge_to_col`	Source/target columns in ‘edge_list_df‘.
`canonical_from_col, canonical_to_col`	From/to columns in ‘canonicals_df‘.
`canonical_duplicate_from_policy`	How to handle a canonical source with multiple distinct targets. See [build_fold_map()].
`canonical_loop_handling`	How to handle canonical cycles. See [build_fold_map()].

Value

The edge list with ‘edge_from_col‘ and ‘edge_to_col‘ folded through canonical declarations. The applied fold map is attached as attribute ‘"fold_map"‘.

See Also

Other edge-list resolvers: `resolve_links()`, `resolve_redirects()`

Examples

```
edges <- data.frame(from = "A", to = "B")
canonicals <- data.frame(from = "B", to = "C")
resolve_canonicals(edges, canonicals)
```

`resolve_canonical_urls` *Resolve URLs through rel=canonical declarations*

Description

Resolves a character vector through declared ‘rel=canonical‘ folds. This is distinct from URL syntax canonicalization; inputs are expected to already be in the same URL namespace as the canonical table.

Usage

```
resolve_canonical_urls(
  urls,
  canonicals_df,
  canonical_from_col = "from",
  canonical_to_col = "to",
  canonical_duplicate_from_policy = c("strict", "first_wins", "last_wins",
    "most_frequent", "prune_source", "resolve_if_consistent"),
  canonical_loop_handling = c("error", "prune_loop", "break_arrow")
)
```

Arguments

`urls`	Character vector of URLs to resolve.
`canonicals_df`	A data frame of declared canonical links.
`canonical_from_col`, `canonical_to_col`	From/to columns in ‘canonicals_df‘.
`canonical_duplicate_from_policy`	How to handle a canonical source with multiple distinct targets. See [build_fold_map()].
`canonical_loop_handling`	How to handle canonical cycles. See [build_fold_map()].

Value

A data frame with ‘original‘, ‘resolved‘, ‘changed‘, and ‘signal‘ columns. The applied fold map is attached as attribute ‘"fold_map"‘.

See Also

Other URL-vector resolvers: `resolve_folded_urls()`, `resolve_redirect_urls()`

Examples

```
canonicals <- data.frame(
  from = c("A", "B"),
  to = c("B", "C")
)
resolve_canonical_urls(c("A", "B", "X"), canonicals)
```

`resolve_folded_urls` *Resolve URLs through composed redirects and canonicals*

Description

Resolves URL vectors with the same composed 3xx redirect plus declared ‘rel=canonical‘ fold-map engine used by [pagerank()] and [build_fold_map()]. This helper performs signal folding only; it does not perform URL syntax canonicalization.

Usage

```
resolve_folded_urls(
  urls,
  redirects_df = NULL,
  canonicals_df = NULL,
  redirect_from_col = "from",
  redirect_to_col = "to",
  canonical_from_col = "from",
  canonical_to_col = "to",
  duplicate_from_policy = c("strict", "first_wins", "last_wins", "most_frequent",
    "prune_source", "resolve_if_consistent"),
  loop_handling = c("error", "prune_loop", "break_arrow"),
  canonical_duplicate_from_policy = c("strict", "first_wins", "last_wins",
    "most_frequent", "prune_source", "resolve_if_consistent"),
  canonical_loop_handling = c("error", "prune_loop", "break_arrow"),
  canonical_conflict_policy = c("redirect_wins", "error", "canonical_wins")
)
```

Arguments

`urls`	Character vector of URLs to resolve.
`redirects_df`	Optional data frame of redirect rules, or ‘NULL‘.
`canonicals_df`	Optional data frame of declared canonical links, or ‘NULL‘.
`redirect_from_col`, `redirect_to_col`	From/to columns in ‘redirects_df‘. Default ‘"from"‘ / ‘"to"‘.
`canonical_from_col`, `canonical_to_col`	From/to columns in ‘canonicals_df‘. Default ‘"from"‘ / ‘"to"‘.
`duplicate_from_policy`	How to handle a redirect source with multiple distinct targets. See [resolve_redirects()]. Default ‘"strict"‘.

`loop_handling` How to handle redirect cycles. See [resolve_redirects()]. Default ‘"error"‘. Also governs cross-signal cycles in the composed graph.

`canonical_duplicate_from_policy`
How to handle a canonical source with multiple distinct declared canonicals. Reuses the ‘duplicate_from_policy‘ enum. Default ‘"strict"‘.

`canonical_loop_handling`
How to handle cycles among declared canonicals. Reuses the ‘loop_handling‘ enum. Default ‘"error"‘.

`canonical_conflict_policy`
How to resolve a redirect-vs-canonical disagreement on the **same source** URL. One of:

- **‘"redirect_wins"‘** (Default) The 3xx redirect wins; a canonical declared on a URL that itself redirects is ignored and flagged, never transferred onto the redirect target.
- **‘"error"‘** Error when a redirect and a canonical disagree on the same source (after audit context is computed). Sources where the two signals agree do not error.
- **‘"canonical_wins"‘** The declared canonical wins for that source; still flagged in the audit. The explicit exception to the default ignored-canonical-on-redirecting-source rule.

Value

A data frame with ‘original‘, ‘resolved‘, ‘changed‘, and ‘signal‘ columns. The exported fold map is attached as attribute ‘"fold_map"‘ and cross-signal audit tables are attached as ‘"conflicts"‘ and ‘"ignored_canonicals"‘.

See Also

Other URL-vector resolvers: `resolve_canonical_urls()`, `resolve_redirect_urls()`

Examples

```
redirects <- data.frame(from = "B", to = "C")
canonicals <- data.frame(from = "A", to = "B")
resolve_folded_urls(c("A", "B", "X"), redirects, canonicals)
```

---

resolve_links *Resolve Links Through Redirects*

Description

Applies redirect rules to an edge list and returns the resolved link graph without computing PageRank. Useful for inspecting what the link graph looks like after redirects are applied, deduplication, and optional URL cleaning.

Usage

```
resolve_links(
  edge_list_df,
  redirects_df = NULL,
  clean_urls = TRUE,
  self_loops = c("drop", "keep"),
  edge_from_col = "from",
  edge_to_col = "to",
  redirect_from_col = "from",
  redirect_to_col = "to",
  duplicate_from_policy = c("strict", "first_wins", "last_wins", "most_frequent",
    "prune_source", "resolve_if_consistent"),
  loop_handling = c("error", "prune_loop", "break_arrow"),
  rurl_params = list()
)
```

Arguments

`edge_list_df`	A data frame representing the edge list with at least two columns for source and target URLs.
`redirects_df`	A data frame containing redirect rules with 'from' and 'to' columns (or as specified by 'redirect_from_col'/'redirect_to_col'). If 'NULL', no redirects are applied.
`clean_urls`	Logical, whether to clean/normalize URLs using `rurl::clean_url` before resolving. Default 'TRUE'.
`self_loops`	Character, how to handle self-loops created after redirect resolution. One of '"drop"' (default) or '"keep"'.
`edge_from_col, edge_to_col`	Names of the from/to columns in 'edge_list_df'. Default '"from"' and '"to"'.
`redirect_from_col, redirect_to_col`	Names of the from/to columns in 'redirects_df'. Default '"from"' and '"to"'.
`duplicate_from_policy`	How to handle conflicting redirects. Passed through to [resolve_redirects()]. Default '"strict"'.
`loop_handling`	How to handle redirect cycles. Passed through to [resolve_redirects()]. Default '"error"'.
`rurl_params`	Named list of additional arguments passed to `rurl::clean_url` when 'clean_urls = TRUE'.

Value

A data frame with the same columns as 'edge_list_df', but with URLs replaced by their final redirect destinations, duplicate edges removed, and self-loops handled according to 'self_loops'.

See Also

Other edge-list resolvers: `resolve_canonicals()`, `resolve_redirects()`

Examples

```
edges <- data.frame(
  from = c("A", "B", "C", "A"),
  to = c("B", "C", "D", "B")
)
redirects <- data.frame(
  from = c("B", "C"),
  to = c("B_final", "C_final")
)
resolve_links(edges, redirects, clean_urls = FALSE)

# Without redirects: just deduplicate and clean
resolve_links(edges, clean_urls = FALSE)

# Inspect the graph before and after a redirect change
before <- resolve_links(edges, clean_urls = FALSE)
new_redirects <- data.frame(
  from = "D", to = "B_final"
)
after <- resolve_links(edges, new_redirects, clean_urls = FALSE)
```

---

resolve_redirects *Resolve Redirects in an Edge List*

Description

Updates an edge list by replacing URLs with their final destinations based on a redirect data frame. Handles redirect chains, detects cycles, and resolves conflicting redirects using configurable policies.

Usage

```
resolve_redirects(
  edge_list_df,
  redirects_df,
  edge_from_col = "from",
  edge_to_col = "to",
  redirect_from_col = "from",
  redirect_to_col = "to",
 duplicate_from_policy = c("strict", "first_wins", "last_wins", "most_frequent",
    "prune_source", "resolve_if_consistent"),
  loop_handling = c("error", "prune_loop", "break_arrow")
)
```

Arguments

edge_list_df	A data frame representing the edge list.

`redirects_df` A data frame containing redirect rules, with 'from' and 'to' columns specifying the source and target of a redirect.

`edge_from_col` Character, the name of the column in 'edge_list_df' containing source URLs. Default "from".

`edge_to_col` Character, the name of the column in 'edge_list_df' containing target URLs. Default "to".

`redirect_from_col`
Character, the name of the column in 'redirects_df' containing source URLs of redirects. Default "from".

`redirect_to_col`
Character, the name of the column in 'redirects_df' containing target URLs of redirects. Default "to".

`duplicate_from_policy`
Character, how to handle conflicting redirects (same source URL mapping to multiple distinct targets). One of:

"strict" (Default) Error on any conflict.

"first_wins" Keep the first occurrence for each conflicting source.

"last_wins" Keep the last occurrence for each conflicting source.

"most_frequent" Keep the most common target. Ties broken by first occurrence.

"prune_source" Remove ALL redirects from any conflicting source.

"resolve_if_consistent" Allow exact duplicates; error only on true conflicts where targets differ.

`loop_handling` Character, how to handle redirect cycles (loops). One of:

"error" (Default) Error when a redirect cycle is detected.

"prune_loop" Remove all edges involved in cycles. URLs in the loop remain unresolved (map to themselves).

"break_arrow" For each cycle, keep the node with the highest in-degree as the sink and remove edges pointing away from it within the cycle. This preserves as much of the chain as possible.

Details

Self-referencing redirects (where from == to) and any redirects with NA in from or to are automatically filtered out before processing.

When crawl data contains conflicting redirects (the same URL redirecting to different targets), use `duplicate_from_policy` to control the behavior. The default `"strict"` preserves backward compatibility by erroring on any conflict.

Redirect resolution uses a graph-based approach: an igraph is built from the redirect rules, strongly connected components (SCCs) are used to detect loops, and the `loop_handling` policy determines what happens to cycles. After loop handling, each URL is mapped to its terminal destination by traversing the acyclic graph.

Value

An updated 'edge_list_df' with URLs in 'edge_from_col' and 'edge_to_col' replaced by their final resolved destinations.

See Also

Other edge-list resolvers: resolve_canonicals(), resolve_links()

Examples

```
edges <- data.frame(
  from = c("A", "B", "C"),
  to = c("B", "C", "D")
)
redirects <- data.frame(
  from = c("B", "C", "E"),
  to = c("B_final", "C_final", "E_final")
)
resolve_redirects(edges, redirects)

# Example with a redirect chain
edges_chain <- data.frame(from = "X", to = "Y")
redirects_chain <- data.frame(
  from = c("Y", "Z"),
  to = c("Z", "Z_final")
)
resolve_redirects(edges_chain, redirects_chain)

# Example with conflicting redirects resolved via first_wins
edges_conflict <- data.frame(
  from = "A", to = "B"
)
redirects_conflict <- data.frame(
  from = c("B", "B"),
  to = c("C", "D")
)
resolve_redirects(edges_conflict, redirects_conflict,
  duplicate_from_policy = "first_wins"
)

# Example with different column names
edges_custom <- data.frame(
  source_url = "Page1", target_url = "Page2"
)
redirects_custom <- data.frame(
  original = "Page2", final = "Page2_resolved"
)
resolve_redirects(edges_custom, redirects_custom,
  edge_from_col = "source_url",
  edge_to_col = "target_url",
  redirect_from_col = "original",
  redirect_to_col = "final"
)
```

`resolve_redirect_urls` *Resolve URLs Through Redirects*

Description

Given a character vector of URLs and a redirect data frame, resolves each URL to its final destination by following redirect chains. Unlike `resolve_redirects`, this function does not require an edge list – it works directly on a list of URLs. This helper is redirect-only; use [resolve_canonical_urls()] for 'rel=canonical' folding or [resolve_folded_urls()] for composed redirect plus canonical folding.

Usage

```
resolve_redirect_urls(
  urls,
  redirects_df,
  redirect_from_col = "from",
  redirect_to_col = "to",
 duplicate_from_policy = c("strict", "first_wins", "last_wins", "most_frequent",
    "prune_source", "resolve_if_consistent"),
  loop_handling = c("error", "prune_loop", "break_arrow")
)
```

Arguments

`urls`	Character vector of URLs to resolve.
`redirects_df`	A data frame containing redirect rules.
`redirect_from_col`	Character, name of the source column. Default `"from"`.
`redirect_to_col`	Character, name of the target column. Default `"to"`.
`duplicate_from_policy`	How to handle conflicting redirects. Passed through to redirect preprocessing. Default `"strict"`.
`loop_handling`	How to handle redirect cycles. Default `"error"`. Use `"prune_loop"` or `"break_arrow"` to resolve despite loops.

Value

A data frame with columns:

original The input URL.

resolved The final destination after following all redirects.

changed Logical, whether the URL was modified by a redirect.

See Also

Other URL-vector resolvers: resolve_canonical_urls(), resolve_folded_urls()

Examples

```
redirects <- data.frame(
  from = c("A", "B", "C"),
  to = c("B", "C", "Final")
)

# Resolve specific URLs
resolve_redirect_urls(c("A", "B", "X"), redirects)

# X is not in the redirect map, so it stays as-is
```

---

salsa *Master SALSA hub/authority calculation wrapper*

Description

Computes Lempel & Moran's SALSA hub and authority scores over the same cleaned, redirect/canonical-folded, domain-filtered, deduplicated link graph that [pagerank()] builds, so node identities line up across the centrality measures. SALSA is a stochastic variant of HITS: it runs HITS-style mutual reinforcement as PageRank-style random walks on the bipartite hub/authority graph, yielding stationary-distribution scores instead of dominant eigenvectors.

Usage

```
salsa(
  edge_list_df,
  redirects_df = NULL,
  clean_edge_urls = TRUE,
  clean_redirect_urls = TRUE,
  rurl_params = list(),
  self_loops = c("drop", "keep"),
  drop_isolates_flag = TRUE,
  duplicate_edge_policy = c("collapse", "aggregate", "count_instances"),
  edge_from_col = "from",
  edge_to_col = "to",
  redirect_from_col = "from",
  redirect_to_col = "to",
 duplicate_from_policy = c("strict", "first_wins", "last_wins", "most_frequent",
    "prune_source", "resolve_if_consistent"),
  loop_handling = c("error", "prune_loop", "break_arrow"),
  canonicals_df = NULL,
  canonical_from_col = "from",
  canonical_to_col = "to",
```

```
  clean_canonical_urls = TRUE,
  canonical_duplicate_from_policy = c("strict", "first_wins", "last_wins",
    "most_frequent", "prune_source", "resolve_if_consistent"),
  canonical_loop_handling = c("error", "prune_loop", "break_arrow"),
  canonical_conflict_policy = c("redirect_wins", "error", "canonical_wins"),
  keep_domains = NULL,
  exclude_domains = NULL,
  keep_hosts = NULL,
  exclude_hosts = NULL,
  ...
)
```

Arguments

edge_list_df	A data frame representing the edge list, typically with columns like "from" and "to". Edges are expected to be page-to-page hyperlinks: ‘pagerank()‘ is graph-agnostic and treats every endpoint as a node, so resource links (images, CSS, JS, and other non-HTML references) must be filtered upstream or they collect authority as ordinary vertices. ‘pagerank_screaming_frog()‘ does this at the crawl boundary via [sf_graph_eligible()] (‘Hyperlink‘ only); a hand-built or non-SF edge list should apply the same hyperlink-only filter before scoring.
redirects_df	An optional data frame for redirect rules, typically with "from" and "to" columns. Defaults to NULL.
clean_edge_urls	Logical, whether to clean URLs in the edge list. Defaults to TRUE.
clean_redirect_urls	Logical, whether to clean URLs in the redirect list. Defaults to TRUE. Only effective if ‘redirects_df‘ is provided.
rurl_params	A list of parameters to pass to ‘rurl::clean_url‘. Defaults to an empty list. ‘protocol_handling‘ defaults to ‘"keep"‘ and ‘case_handling‘ to ‘"lower_host"‘ for cross-project canonicalization consistency. If you set ‘host_encoding‘ (‘"idna"‘ or ‘"unicode"‘) to fold internationalized (IDN) hosts, that same value is also passed to the domain-filtering step so its comparisons stay consistent with the cleaned node keys. (Registrable-domain matching is encoding-independent, so this only matters if host-level filtering is involved.)
self_loops	A character string specifying how to handle self-loops. Either "drop" (default) or "keep".
drop_isolates_flag	Logical, whether to drop isolated nodes before PageRank computation. Defaults to TRUE.
duplicate_edge_policy	How repeated ‘from -> to‘ rows are represented after URL cleaning, redirect/canonical folding, and domain filtering. One of: **‘"collapse"‘** (default) Destination-level surfer: repeated rows collapse to one unweighted destination edge, preserving legacy ‘get_unique_edges()‘ behavior and the common binary PageRank convention.

‘"aggregate"‘ Collapse each ‘from -> to‘ pair with [aggregate_edges()] semantics. Numeric columns, including ‘weight_col‘, are summed; logical columns such as ‘nofollow‘ use the default ‘"any"‘ conflict policy.

‘"count_instances"‘ Link-slot / edge-level surfer: repeated rows increase transition probability. With no ‘weight_col‘, each surviving ‘from -> to‘ pair receives an internal weight equal to its duplicate-row count. With ‘weight_col‘, weights are summed and an ‘instance_count‘ audit column is retained.

`edge_from_col`, `edge_to_col`
: Names of from/to columns in ‘edge_list_df‘.

`redirect_from_col`, `redirect_to_col`
: Names of from/to columns in ‘redirects_df‘.

`duplicate_from_policy`
: How to handle conflicting redirects in ‘redirects_df‘. Passed through to [resolve_redirects()]. Default ‘"strict"‘ (error on conflicts). See [resolve_redirects()] for all available policies.

`loop_handling`
: How to handle redirect cycles. Passed through to [resolve_redirects()]. Default ‘"error"‘. See [resolve_redirects()] for all available policies.

`canonicals_df`
: An optional data frame of declared ‘rel=canonical‘ links, with ‘from‘/‘to‘ columns (or as set by ‘canonical_from_col‘ / ‘canonical_to_col‘) pairing a source URL with the canonical it declares. Default ‘NULL‘ (opt-in; the default preserves current behavior). Canonicals are a **distinct, advisory** signal from enforced 3xx ‘redirects_df‘: they are tracked separately and audited via [audit_canonicals()] / [audit_fold()], then folded into the same composed map as redirects (see [build_fold_map()]). Self-canonicals drop as no-ops.

`canonical_from_col`, `canonical_to_col`
: From/to columns in ‘canonicals_df‘. Default ‘"from"‘ / ‘"to"‘.

`clean_canonical_urls`
: Logical, whether to clean URLs in ‘canonicals_df‘ using the same resolved ‘rurl_params‘ profile as edge and redirect cleaning. Default ‘TRUE‘. Only effective when ‘canonicals_df‘ is provided.

`canonical_duplicate_from_policy`
: How to handle a canonical source that declares multiple distinct canonicals. Reuses the ‘duplicate_from_policy‘ enum (see [resolve_redirects()]). Default ‘"strict"‘.

`canonical_loop_handling`
: How to handle cycles among declared canonicals. Reuses the ‘loop_handling‘ enum. Default ‘"error"‘.

`canonical_conflict_policy`
: How to resolve a redirect-vs-canonical disagreement on the **same source** URL. One of ‘"redirect_wins"‘ (default; the 3xx wins and the canonical on a redirecting source is ignored and flagged), ‘"error"‘ (error on genuine disagreement), or ‘"canonical_wins"‘ (the declared canonical wins for that source, still flagged). See [build_fold_map()].

`keep_domains`
: Optional character vector of domains to keep. When provided, edges are filtered via [filter_links_by_domain()] so that only links where both endpoints belong to one of the specified domains are included. Useful for restricting to internal links. Default ‘NULL‘ (no domain filtering).

Ordering: filtering runs *after* redirect/canonical folding, so it scopes the post-fold (canonical) namespace, not the crawled input. If an out-of-scope canonical/redirect rewrites the crawled domain onto a different one, filtering on the crawled domain matches nothing (an empty graph). To scope the INPUT you crawled, run [filter_links_by_domain()] on the edge list *before* calling ‘pagerank()‘.

`exclude_domains`
: Optional character vector of domains to exclude. Edges where either endpoint belongs to one of these domains are removed. Like ‘keep_domains‘, this filters the post-fold namespace (see the ordering note above). Default ‘NULL‘ (no exclusion).

`keep_hosts`
: Optional character vector of exact hosts to keep (e.g. ‘"www.example.com"‘), as opposed to registrable domains. Matched on the exact host using the same canonicalization profile as cleaning, so IDN folding (‘host_encoding‘ in ‘rurl_params‘) applies consistently. Default ‘NULL‘.

`exclude_hosts`
: Optional character vector of exact hosts to exclude. Edges where either endpoint matches one of these hosts are removed. Ignore rules override keep rules. Default ‘NULL‘.

`...`
: Additional arguments forwarded to [compute_salsa()].

Details

Relationship to the PageRank pipeline

‘salsa()‘ reuses the exact identity-forming steps of [pagerank()] — URL canonicalization (the same resolved ‘rurl‘ profile), the same composed redirect + canonical fold map, the same domain/host filtering, the same ‘duplicate_edge_policy‘ deduplication, and the same self-loop / isolate handling. The resulting vertex set therefore matches ‘pagerank()‘ run with the same arguments, so hub, authority, and PageRank can be joined on ‘node_name‘ without re-canonicalizing.

The PageRank-specific, *forward-flow* modeling devices have **no SALSA analogue and are intentionally not exposed**: nofollow evaporation, the indexability (noindex / robots.txt) transforms, the TIPR teleport prior, and the ‘reverse‘ flag. SALSA already computes both directions of authority flow (hub is the outflow-oriented score, authority the inflow-oriented one).

SALSA versus HITS, and the pagerankr adaptation

Where [hits()] takes the dominant eigenvectors of $A^\top A$ and $AA^\top$, SALSA replaces the mutual-reinforcement iteration with two *stochastic* Markov chains on the bipartite hub/authority graph; their stationary distributions are the scores, which on a connected graph reduce to a simple in-/out-degree closed form (see [compute_salsa()]). Because the chains are stochastic, SALSA is far less prone than HITS to the "tightly-knit community" effect, where a dense cluster of mutually linking pages dominates the top scores.

Lempel & Moran’s original SALSA ran on a query-focused base set. ‘salsa()‘ instead runs on the **full (or user-filtered) site graph** that ‘pagerankr‘ assembles — a documented site-graph adaptation of the focused-subgraph algorithm. Treat the scores as site-wide structural centralities, not query-relevance scores. Coverage differs from PageRank: a node’s ‘hub‘ is ‘NA‘ when it has no outlinks and its ‘authority‘ is ‘NA‘ when it has no inlinks (see [compute_salsa()]).

Value

A data frame with one row per node and columns ‘node_name‘, ‘hub‘, and ‘authority‘ (column names configurable via ‘...‘). Hub and authority each sum to ‘1‘ over their non-‘NA‘ entries.

References

Lempel, R. & Moran, S. (2001). SALSA: The Stochastic Approach for Link-Structure Analysis. *ACM Transactions on Information Systems*, 19(2), 131-160.

See Also

[compute_salsa()] for the computational core, [hits()] for the HITS analogue, and [pagerank()] for the PageRank analogue sharing this identity pipeline.

Examples

```
edges <- data.frame(
  from = c("http://A.com/", "http://A.com/", "B.com"),
  to = c("B.com", "C.com", "C.com")
)
salsa(edges)

# Hub vs authority: a pure outflow page tops hub, a pure inflow page tops
# authority.
s <- salsa(edges)
s[which.max(s$hub), ]
s[which.max(s$authority), ]
```

---

screaming_frog_bundle *Compose Screaming Frog node and link exports*

Description

Builds the stable ‘screaming_frog_bundle‘ handoff object from an **Internal: All** export and one **All Inlinks** or **All Outlinks** export. The component adapters remain the source of truth: raw link observations stay separate from graph-eligible edges, and node, redirect, canonical, and indexability tables are exposed unchanged for downstream scoring.

‘pagerankr‘ accepts a Screaming Frog **Internal: All** export as node metadata and one **All Inlinks** or **All Outlinks** bulk export as link observations. Internal: All is never treated as an edge list. Both link exports use ‘Source -> Destination‘; the export kind records provenance and never changes orientation.

Usage

```
screaming_frog_bundle(
  internal,
  links,
  link_export_kind = c("all_inlinks", "all_outlinks"),
```

```
  origin_policy = c("all", "html", "rendered"),
  endpoint_action = c("drop", "error")
)
```

Arguments

`internal`	A path/data frame accepted by [screaming_frog_internal()], or an existing 'screaming_frog_internal' object.
`links`	A path/data frame accepted by [screaming_frog_links()], or an existing 'screaming_frog_links' object.
`link_export_kind`	Declared link export kind when 'links' is not already a 'screaming_frog_links' object: '"all_inlinks"' or '"all_outlinks"'.
`origin_policy, endpoint_action`	Passed to [screaming_frog_links()] when 'links' is not already imported.

Value

An S3 object of class 'screaming_frog_bundle' with stable top-level fields 'nodes', 'observations', 'edges', 'redirects', 'canonicals', 'indexability', 'diagnostics', and 'provenance'.

Input boundary

Future Screaming Frog adapters accept either a file path or a data frame. Files are CSV, may contain a UTF-8 byte-order mark, and are read by first inspecting the header and then selecting only contract columns. This keeps 500+ MB link exports bounded to the columns needed by the requested adapter. Data-frame inputs follow the same alias resolution and column ordering. Unknown extra columns are ignored and reported; missing required columns fail with their normalized contract names.

Raw URLs are preserved at this boundary. URL cleaning belongs to the scoring path and is performed once with 'pagerankr''s pinned 'rurl' canonicalization profile.

Link observation semantics

Raw link observations and graph-eligible edges are separate tables. Duplicate observations are preserved. Only rows whose 'type' is '"Hyperlink"' are graph-eligible by default. Resource, sitemap, hreflang, redirect, and canonical observations are not silently promoted to PageRank edges; redirects and canonicals remain dedicated signals.

'follow' is the primary nofollow field. 'rel' is retained and parsed independently for diagnostics, so disagreements can be reported. Link position is normalized only for 'Navigation', 'Content', 'Footer', 'Header', and 'Aside'; 'Head', blanks, and unknown values remain unmapped rather than being guessed. Link origin and link path are preserved as provenance. Link path is an XPath-like source locator, not a URL path.

Stable bundle shape

The public 'screaming_frog_bundle' object introduced by the adapter tickets is an S3 list with these stable top-level fields, in order:

nodes Normalized Internal: All node facts.

observations Lossless normalized link observations.

edges The graph-eligible ‘from‘ / ‘to‘ subset.

redirects ‘from‘ / ‘to‘ redirect signals.

canonicals ‘from‘ / ‘to‘ canonical signals.

indexability URL-level indexability facts.

diagnostics Counts, omissions, invalid values, and disagreements.

provenance Input kind, source, schema aliases, and schema clues.

This topic freezes the contract consumed by the SF1-SF5 implementation tickets. The complete object is constructed by [screaming_frog_bundle()].

Examples

```
internal <- data.frame(
  Address = c("https://example.com/", "https://example.com/a"),
  `Status Code` = c("200", "200"),
  check.names = FALSE
)
links <- data.frame(
  Type = "Hyperlink",
  Source = "https://example.com/",
  Destination = "https://example.com/a",
  Follow = "TRUE",
  check.names = FALSE
)
bundle <- screaming_frog_bundle(internal, links, "all_outlinks")
bundle$edges
```

`screaming_frog_internal` *Import Screaming Frog Internal: All node facts*

Description

Normalizes a Screaming Frog **Internal: All** CSV or data frame into node, redirect, canonical, and indexability tables. This is a node-side adapter: it does not reconstruct links from aggregate Inlinks/Outlinks counts.

Usage

```
screaming_frog_internal(x)
```

Arguments

`x`	A path to an Internal: All CSV file or an equivalent data frame.

Details

URLs are preserved as exported. Redirects are emitted only for valid 3xx rows with a non-blank destination. Canonicals are derived independently, including self-canonicals for audit.

Value

An object of class ‘screaming_frog_internal‘ with components:

nodes Normalized node facts in input row order. Columns are ‘url‘, ‘segments‘, ‘content_type‘, ‘http_status‘, ‘status‘, ‘indexability‘, ‘indexability_status‘, ‘canonical‘, ‘redirect_to‘, ‘redirect_type‘, ‘crawl_allowed‘, ‘indexing_allowed‘, robots fields, language/timestamps, and selected crawl metrics. Optional absent fields are typed ‘NA‘ columns.

redirects Raw ‘from‘ / ‘to‘ redirect pairs from valid 3xx rows.

canonicals Raw ‘from‘ / ‘to‘ canonical pairs, including self-canonicals.

indexability URL-level facts compatible with ‘pagerank()‘’s indexability input.

diagnostics Deterministic counts, missing optional and ignored columns, duplicate addresses, and row-level structural issues.

provenance Input identity, retained input-row IDs, and the normalized-to-detected column manifest.

Examples

```
internal <- data.frame(
  Address = c("https://example.com/", "https://example.com/old"),
  `Status Code` = c("200", "301"),
  `Redirect URL` = c("", "https://example.com/new"),
  check.names = FALSE
)
imported <- screaming_frog_internal(internal)
imported$nodes
imported$redirects
```

---

screaming_frog_links *Import Screaming Frog All Inlinks or All Outlinks observations*

Description

Normalizes a Screaming Frog **All Inlinks** or **All Outlinks** CSV or data frame. Both exports retain their native ‘Source -> Destination‘ orientation. Raw observations are kept separately from graph-eligible edges, and URLs are preserved for one downstream canonicalization pass.

Usage

```
screaming_frog_links(
  x,
  export_kind = c("all_inlinks", "all_outlinks"),
  origin_policy = c("all", "html", "rendered"),
  endpoint_action = c("drop", "error")
)
```

Arguments

x	A path to an All Inlinks/All Outlinks CSV file or an equivalent data frame.
export_kind	Declared export provenance: ‘"all_inlinks"‘ or ‘"all_outlinks"‘. This never changes edge orientation.
origin_policy	Which DOM observations may become graph edges: ‘"all"‘ (default), ‘"html"‘, or ‘"rendered"‘. Combined ‘"HTML & Rendered HTML"‘ observations qualify for either selective policy. The raw observation table is never filtered.
endpoint_action	How graph-eligible rows with a blank source or destination are handled: ‘"drop"‘ (default) or ‘"error"‘. Such rows remain in ‘observations‘ in either mode.

Details

Only ‘Type = "Hyperlink"‘ observations are graph-eligible. ‘Follow‘ is the authoritative field used to derive ‘nofollow‘; ‘Rel‘ is parsed independently for disagreement diagnostics. Duplicate observations are not aggregated.

Value

An object of class ‘screaming_frog_links‘ with components:

observations Normalized observations in input row order.

edges Unaggregated, graph-eligible ‘from‘ / ‘to‘ rows with nofollow, placement, origin, and link provenance. ‘position_index‘ carries each link’s reading-order rank among its source page’s content links (‘1‘ = first), materialized from document order for ‘all_outlinks‘ and ‘NA‘ otherwise; it feeds [pagerank()]’s ‘position_col‘.

diagnostics Input, eligibility, endpoint, type, origin, Follow/Rel, and schema counts plus row-level issues.

provenance Export kind, source, policy, retained input-row IDs, and detected columns.

Examples

```
links <- data.frame(
  Type = c("Hyperlink", "Image"),
  Source = c("https://example.com/", "https://example.com/"),
  Destination = c("https://example.com/a", "https://example.com/logo.png"),
  Follow = c("TRUE", "TRUE"),
  check.names = FALSE
)
imported <- screaming_frog_links(links, "all_outlinks")
imported$edges
```

seed_prior *Seed Teleport Prior for Personalized PageRank*

Description

Build a teleport prior (a ‘prior_df‘) concentrated on a set of **seed** pages. A seed prior is the teleportation vector that biases the random surfer toward the seeds instead of jumping uniformly, and it is the single ingredient shared by the seed-biased members of the PageRank family: [trustrank()] (trusted seeds) and [topic_feeder_pagerank()] (a target cluster) both build one and hand it to [pagerank()].

‘seed_prior()‘ is **orientation-agnostic**: it does nothing but turn a seed set into a ‘url‘/‘weight‘ prior. Whether teleport mass then flows *outward* from the seeds (trust) or is accumulated by pages that *point into* the seeds (feeders) is a property of the **graph**, chosen by the caller — not of this builder. [trustrank()] runs the prior on the forward graph; [topic_feeder_pagerank()] runs the identical prior on the reversed graph (‘pagerank(reverse = TRUE)‘). That is precisely why one builder serves both: the direction lives in the wrapper, not in the prior.

For the multi-topic case ([topic_sensitive_pagerank()]) the prior is built internally per topic from a named list; ‘seed_prior()‘ covers the single-seed-set case that the two convenience wrappers share.

Usage

```
seed_prior(
  seeds,
  seed_weight = NULL,
  seed_url_col = "url",
  seed_weight_col = "weight"
)
```

Arguments

seeds	The seed set. Either a character vector of seed URLs (each gets equal weight unless ‘seed_weight‘ is given), or a data frame with a URL column and a numeric weight column (see ‘seed_url_col‘ / ‘seed_weight_col‘) for unequal emphasis.
seed_weight	Optional numeric weight for a character-vector ‘seeds‘: either one value per seed or a single value recycled to all seeds. Ignored when ‘seeds‘ is a data frame. Default ‘NULL‘ (every seed weight ‘1‘, i.e. a uniform distribution over the seed set).
seed_url_col, seed_weight_col	Column names used when ‘seeds‘ is a data frame. Defaults ‘"url"‘ / ‘"weight"‘. Ignored for a character vector.

Details

Seed weights are an **additive teleport budget**: when two seed URLs fold onto the same vertex (redirect / canonical variants) their weights sum, exactly as the [pagerank()] / [align_prior_to_vertices()] prior contract specifies. Equal weights give a uniform distribution over the seed set; unequal

weights express graded emphasis (graded trust for [trustrank()], graded cluster importance for [topic_feeder_pagerank()]).

Value

A data frame with ‘url‘ and ‘weight‘ columns, suitable as the ‘prior_df‘ argument to [pagerank()].

See Also

[trustrank()], [topic_feeder_pagerank()], [topic_sensitive_pagerank()], [pagerank()], [align_prior_to_vertices()]

Examples

```
# A trusted-seed prior for TrustRank: run on the FORWARD graph, trust flows
# outward from the seeds.
prior <- seed_prior(c("/", "/hub"))
prior

edges <- data.frame(
  from = c("/", "/hub", "/feeder"),
  to = c("/hub", "/ai", "/ai")
)
pagerank(edges, prior_df = prior, clean_edge_urls = FALSE)

# The SAME builder makes a cluster prior for feeder PageRank; the only
# difference is the graph orientation you run it on (reverse = TRUE).
cluster <- seed_prior("/ai")
pagerank(edges, prior_df = cluster, reverse = TRUE, clean_edge_urls = FALSE)

# Graded emphasis via a data frame.
seed_prior(data.frame(url = c("/a", "/b"), weight = c(3, 1)))
```

---

sf_container_from_path

Derive a link's container component from its DOM path

Description

Reduces a Screaming Frog `Link Path` to the **component the link sits in**, stable across every page that component appears on. This is the identity the boilerplate detector conditions on: see `pagerank()`'s `container_col`.

Two steps:

1. **Strip numeric predicates, keep class predicates.** Screaming Frog's `Link Path` is a hybrid, using `[@class='...']` where classes exist and positional `[n]` elsewhere. Positions are unstable — the same recycled call-to-action lands at `p[5]` on a post with four preceding paragraphs and `p[3]` on a shorter one — while a class is exactly the stable component identifier we want.
2. **Drop the trailing <a> step**, whatever predicate it carries. The anchor's own class describes the link, not the component containing it.

Note this cuts the **opposite** way from `sf_region_from_path()`, which strips class predicates so that a `div[@class='site-footer']` is not mistaken for a `<footer>`. The two answer different questions — *which region is this* versus *is this the same component* — and the inconsistency is deliberate.

Usage

```
sf_container_from_path(x)
```

Arguments

x	A vector (typically character) of Screaming Frog link paths, e.g. `"//body/main/article/p[5]/a[1]"`.

Value

A character vector the same length as `x` holding the container path. Blank strings and `NA` yield `NA`, leaving those rows unscored by the detector.

See Also

[pagerank()], whose 'container_col' consumes the result.

Other Screaming Frog toolkit: `sf_contract()`, `sf_graph_eligible()`, `sf_normalize_position()`, `sf_parse_follow()`, `sf_read_input()`, `sf_region_from_path()`, `sf_rel_nofollow()`

Examples

```
sf_container_from_path(c(
  "//body/main/article/p[5]/a[1]", # positions stripped
  "//body/main/article/p[3]/a[1]", # ... so these two agree
  "//body/div[@class='cta']/a", # class kept as the component identity
  "//body/div[@class='cta']/a[@class='btn']" # anchor's own class dropped
))
```

`sf_contract` *Screaming Frog import contract*

Description

Returns the frozen contract that governs how Screaming Frog exports are read and normalized: the accepted export kinds, the column schemas (canonical field order, required fields, and header aliases) for the Internal and Inlinks/Outlinks exports, and which link types count as graph-eligible. Inspect it to see exactly which Screaming Frog column headers are recognized before importing a crawl.

Usage

```
sf_contract()
```

Value

A list with components:

version Integer contract version.

bundle_fields Character vector of the fields present on a `screaming_frog_bundle()` object.

export_kinds Character vector of accepted export kinds, used by `sf_read_input()`.

graph_eligible_types Link types treated as graph edges; see `sf_graph_eligible()`.

internal, links Schemas for the Internal and Inlinks/Outlinks exports, each a list of `order`, `required`, and `aliases`.

See Also

Other Screaming Frog toolkit: `sf_container_from_path()`, `sf_graph_eligible()`, `sf_normalize_position()`, `sf_parse_follow()`, `sf_read_input()`, `sf_region_from_path()`, `sf_rel_nofollow()`

Examples

```
contract <- sf_contract()
contract$export_kinds
contract$graph_eligible_types

# Which Screaming Frog headers map onto the `address` field?
contract$internal$aliases$address
```

---

sf_graph_eligible *Test whether a link type is graph-eligible*

Description

Reports which Screaming Frog link types count as edges in the link graph. Only true hyperlinks build the graph; resource references such as images, stylesheets, and scripts are excluded. The eligible set is `sf_contract()$graph_eligible_types`.

Usage

```
sf_graph_eligible(type)
```

Arguments

`type`	A vector (typically character) of Screaming Frog link types, e.g. `"Hyperlink"` or `"Image"`. Whitespace is trimmed.

Value

A logical vector the same length as `type`, `TRUE` where the type is graph-eligible.

See Also

Other Screaming Frog toolkit: sf_container_from_path(), sf_contract(), sf_normalize_position(), sf_parse_follow(), sf_read_input(), sf_region_from_path(), sf_rel_nofollow()

Examples

```
sf_graph_eligible(c("Hyperlink", "Image", "Stylesheet"))
```

`sf_normalize_position` *Normalize a Screaming Frog link position*

Description

Maps Screaming Frog's "Link Position" values onto the compact vocabulary pagerankr uses for placement-aware weighting: `navigation` becomes `"nav"`, while `header`, `footer`, `aside`, and `content` pass through unchanged. Matching is case-insensitive and whitespace is trimmed. The result is what [pagerank()] consumes through its `placement_col` argument.

Usage

```
sf_normalize_position(x)
```

Arguments

`x`	A vector (typically character) of link positions.

Value

A character vector the same length as `x` containing `"nav"`, `"header"`, `"footer"`, `"aside"`, or `"content"`. Blank strings, `NA`, and unrecognized values yield `NA`.

See Also

Other Screaming Frog toolkit: sf_container_from_path(), sf_contract(), sf_graph_eligible(), sf_parse_follow(), sf_read_input(), sf_region_from_path(), sf_rel_nofollow()

Examples

```
sf_normalize_position(c("Navigation", "Aside", "Content", "", NA))
```

sf_parse_follow *Parse a Screaming Frog follow flag to logical*

Description

Converts the values Screaming Frog writes in a "Follow" column into a logical vector. Matching is case-insensitive and whitespace is trimmed.

Usage

```
sf_parse_follow(x)
```

Arguments

x	A vector (typically character) of follow flags. "true", "yes", "1", and "follow" become TRUE; "false", "no", "0", and "nofollow" become FALSE.

Value

A logical vector the same length as x. Blank strings, NA, and unrecognized values yield NA.

See Also

Other Screaming Frog toolkit: sf_container_from_path(), sf_contract(), sf_graph_eligible(), sf_normalize_position(), sf_read_input(), sf_region_from_path(), sf_rel_nofollow()

Examples

```
sf_parse_follow(c("True", "nofollow", "yes", "", NA))
```

sf_read_input *Read a Screaming Frog export into a normalized data frame*

Description

Reads a Screaming Frog export – either an in-memory data frame or a path to a CSV/Excel file – and returns it with canonical snake_case column names, validated against the schema for export_kind. Header aliases are resolved (e.g. "Address", "URL", and "URI" all map to address), empty strings become NA, and character columns are trimmed.

Usage

```
sf_read_input(x, export_kind, fields = NULL)
```

Arguments

x	A data frame, or a path to a Screaming Frog CSV/Excel export.
export_kind	Character, which export is being read. One of sf_contract()$export_kinds: "internal_all", "all_inlinks", or "all_outlinks".
fields	Optional character vector of additional (non-required) fields to retain beyond the schema's required set. Default NULL keeps the schema's standard field order.

Value

A data frame with canonical snake_case columns. The resolved schema is attached as the "sf_schema" attribute, a list of export_kind, columns, aliases, and ignored_columns.

See Also

Other Screaming Frog toolkit: sf_container_from_path(), sf_contract(), sf_graph_eligible(), sf_normalize_position(), sf_parse_follow(), sf_region_from_path(), sf_rel_nofollow()

Examples

```
crawl <- data.frame(
  Address = c("https://example.com/", "https://example.com/a"),
  `Status Code` = c(200, 200),
  check.names = FALSE
)
out <- sf_read_input(crawl, "internal_all")
names(out)
attr(out, "sf_schema")$export_kind
```

---

sf_region_from_path *Derive a link's page region from its DOM path*

Description

Reads the page region a link sits in out of Screaming Frog's Link Path (an XPath-like source locator), returning the same compact vocabulary as sf_normalize_position(). This is the preferred source of placement, because Link Position loses the enclosing region whenever a <nav> is nested inside one.

Usage

```
sf_region_from_path(x)
```

Arguments

x	A vector (typically character) of Screaming Frog link paths, e.g. "//body/footer/nav/ul/li[1]/a".

Details

The region is the **outermost** layout container on the path — `header`, `footer`, or `aside` — and `"nav"` applies only when the link sits in a `<nav>` that is not inside one of those. So a footer nav resolves to `"footer"`, a header nav to `"header"`, and a standalone nav to `"nav"`. Anything else is `"content"`, which is an acknowledged residual bucket rather than a positive claim about the markup.

Why not just read `Link Position`? On a site whose footer is marked up as `footer > nav > a`, Screaming Frog reports every footer link as `Navigation` and emits no `Footer` bucket at all, so `footer` is not merely mislabeled but unreachable — a user wanting footer at 0.05 and nav at 0.2 has no way to express it. Other sites do emit `Footer`, so the vocabulary silently varies with the site's markup. The DOM path has the region unambiguously in both cases.

Element names are matched on their own: predicates are stripped first, so a `div[@class='site-footer']` is *not* read as a footer. Only real `<footer>` elements are.

Value

A character vector the same length as `x` containing `"nav"`, `"header"`, `"footer"`, `"aside"`, or `"content"`. Blank strings and `NA` yield `NA`, so a caller can fall back to `sf_normalize_position()`.

See Also

[pagerank()], whose 'placement_col' consumes the result.

Other Screaming Frog toolkit: `sf_container_from_path()`, `sf_contract()`, `sf_graph_eligible()`, `sf_normalize_position()`, `sf_parse_follow()`, `sf_read_input()`, `sf_rel_nofollow()`

Examples

```
sf_region_from_path(c(
  "//body/footer/nav/ul/li[1]/a", # footer nav -> footer, not nav
  "//body/header/nav/ul/li[2]/a", # header nav -> header
  "//body/nav/ul/li[1]/a", # standalone nav -> nav
  "//body/main/article/p[5]/a[1]", # -> content
  "//body/div[@class='site-footer']/a" # a class is not an element
))
```

`sf_rel_nofollow` *Detect 'nofollow' in a rel attribute*

Description

Tests whether each value of a link's `rel` attribute contains the `nofollow` token. Values are lowercased and split on commas and whitespace, so `"ugc nofollow"` and `"nofollow,ugc"` both count.

Usage

```
sf_rel_nofollow(x)
```

Arguments

x	A vector (typically character) of rel attribute values.

Value

A logical vector the same length as x: TRUE when the nofollow token is present, FALSE when it is not, and NA for blank strings or NA input.

See Also

Other Screaming Frog toolkit: sf_container_from_path(), sf_contract(), sf_graph_eligible(), sf_normalize_position(), sf_parse_follow(), sf_read_input(), sf_region_from_path()

Examples

```
sf_rel_nofollow(c("nofollow", "ugc nofollow", "sponsored", "", NA))
```

---

simulate_changes *Simulate the PageRank Impact of Link and Redirect Changes*

Description

Compares PageRank before and after proposed changes to the link graph, at both the edge level (adding/removing links) and the URL level (retiring a page behind a redirect, or repointing an existing redirect). The whole graph is recomputed and a before/after table is returned; interpretation is left to the caller. This is a faithful recompute primitive, not a ranking or target-optimization engine.

Usage

```
simulate_changes(
  edge_list_df,
  add_links_df = NULL,
  remove_links_df = NULL,
  redirect_urls_df = NULL,
  remove_urls = NULL,
  redirects_df = NULL,
  on_unknown_target = c("warn", "error", "allow"),
  ...,
  edge_from_col = "from",
  edge_to_col = "to",
  redirect_from_col = "from",
  redirect_to_col = "to",
  label_baseline = "baseline",
  label_proposed = "proposed"
)
```

Arguments

edge_list_df	A data frame representing the current link edge list.
add_links_df	Optional data frame of links to add. Must have the same from/to column names as edge_list_df. Columns present in edge_list_df but absent here are padded with NA on the added rows, so weighted / annotated edge lists keep their schema. Default NULL.
remove_links_df	Optional data frame of links to remove. Matching is by exact from+to pair. Must have the same from/to column names as edge_list_df. Default NULL.
redirect_urls_df	Optional two-column from/to data frame of URL-level redirects to model. Each row retires the from URL and sends its inbound authority to to at 100% pass-through. Retire semantics: the live source's own outbound links are **stripped** before folding (an honest 301 has no body), so the target inherits the source's inbound authority only, never its outlinks. A row for source A **overrides** any prior redirect for A – whether from an earlier row or from the baseline crawl's real 3xx – so "change A into a redirect to C" is a single override. A duplicate source mapping to two distinct targets in one changeset is an error (strict). Default NULL.
remove_urls	Optional character vector of URLs to model as removed (turned into HTTP 404s). Each removed URL keeps its inbound links – other pages still point at it – but now they flow into a dead page: authority arrives and **evaporates** to the shared waste sink rather than redistributing across the site (dangle) or self-amplifying (self-loop). The page's own outbound links are dropped. The node stays in the output holding the mass it absorbed once, flagged "removed-dead" in node_status so its residual score is never misread as earned authority. Under the hood this forces a status_df entry (HTTP 404) into the proposed model only; 4xx and 5xx are one class (no split). A URL appearing in both remove_urls and redirect_urls_df is an error (a page cannot be both a 301 and a 404). To also model cleaning up the inbound links, compose with remove_links_df. Default NULL.
redirects_df	Optional data frame of existing redirects (baseline). Default NULL.
on_unknown_target	How to treat a redirect or link *target* that is not a node in the current graph (it may be a legitimate new page, modeled as a new node that carries inbound authority with no outlinks yet). One of "warn" (default, warn and proceed), "error", or "allow" (proceed silently).
...	Additional arguments passed to both pagerank() calls (e.g., clean_edge_urls, damping, nofollow_col, indexability_df, etc.).
edge_from_col	Name of the from column in edge list data frames. Default "from".
edge_to_col	Name of the to column in edge list data frames. Default "to".
redirect_from_col	Name of the source column in redirect_urls_df and redirects_df. Default "from".

`redirect_to_col`
: Name of the target column in `redirect_urls_df` and `redirects_df`. Default `"to"`.

`label_baseline`
: Label for the baseline model in the comparison output. Default `"baseline"`.

`label_proposed`
: Label for the proposed model in the comparison output. Default `"proposed"`.

Value

The output of `compare_pagerank` (per-node deltas, percentage changes, and rank changes between baseline and proposed) with an added `node_status` column: `"normal"` for a node present and live in both models, `"new-target"` for a node introduced by the changeset (present in the proposed model, absent from the baseline), or `"removed-dead"` for a node retired via `remove_urls` (its proposed score is residual absorbed mass on the way to the waste sink, not earned authority).

`node_status` describes a node's *role in this before/after comparison*, which is a different axis from the `page_state` column `pagerank` attaches to describe a page's *health/index state* (`live` / `noindex` / `robots_blocked` / `response_dead`). The two are deliberately not merged into one vocabulary: `new-target` has no health analogue, and `removed-dead` is the single value bridging both axes — a node whose comparison role is "removed" *because* its proposed health state is `response_dead` (a forced 404). Attributes:

summary Aggregate statistics from `compare_pagerank()`.

proposed The full proposed `pagerank()` result, including its `transition_audit` attribute, so the evaporated-mass cost of a removal is surfaced by default.

manifest A named list describing the changeset: redirects applied, which sources overrode a prior redirect, URLs removed, link add/remove counts, and any unknown targets.

See Also

`simulate_changes_screaming_frog` for the Screaming Frog bundle entry point.

Examples

```
# Current site links
edges <- data.frame(
  from = c("Home", "Home", "About", "Blog"),
  to = c("About", "Blog", "Home", "Home")
)

# Propose adding a link from Blog to About
new_links <- data.frame(
  from = "Blog", to = "About"
)
result <- simulate_changes(edges,
  add_links_df = new_links,
  clean_edge_urls = FALSE
)
print(result)
attr(result, "summary")

# Retire the About page behind a redirect to Home
```

```
retire <- data.frame(from = "About", to = "Home")
simulate_changes(edges, redirect_urls_df = retire, clean_edge_urls = FALSE)

# Model the About page 404-ing: inbound authority flows in and evaporates
simulate_changes(edges, remove_urls = "About", clean_edge_urls = FALSE)
```

`simulate_changes_screaming_frog` *Simulate PageRank Changes on a Screaming Frog Bundle*

Description

The Screaming Frog entry point for [simulate_changes()]. A thin wrapper that mirrors [pagerank_screaming_frog()]: it reuses the same bundle -> [pagerank()] adapter to build the baseline inputs (edges, redirects, canonicals, indexability, placement, nofollow, ...), applies the same URL- and edge-level verbs, and delegates to the shared changeset engine. There is no Screaming Frog-specific simulation logic; both entry points funnel into one engine, so CSV and bundle users get identical what-if capability. The what-if's modeled redirects compose on top of the bundle's real crawled redirects (a changeset redirect for a source wins).

Usage

```
simulate_changes_screaming_frog(
  bundle,
  add_links_df = NULL,
  remove_links_df = NULL,
  redirect_urls_df = NULL,
  remove_urls = NULL,
  on_unknown_target = c("warn", "error", "allow"),
  accepted_placements = NULL,
  link_origins = NULL,
  placement_weights = NULL,
  weight_col = NULL,
  apply_canonicals = TRUE,
  apply_redirects = TRUE,
  preset = NULL,
  ...,
  label_baseline = "baseline",
  label_proposed = "proposed"
)
```

Arguments

`bundle`	A 'screaming_frog_bundle' object.
`add_links_df`, `remove_links_df`, `redirect_urls_df`, `remove_urls`	The changeset verbs. See [simulate_changes()] for their semantics. Link and redirect endpoints (and 'remove_urls') match the bundle's raw crawled URLs

(the ‘from‘/‘to‘ node identities before any folding), using ‘from‘/‘to‘ columns. A modeled ‘remove_urls‘ 404 composes on top of the bundle's real crawled status.

`on_unknown_target`
: How to treat a redirect or link target absent from the current graph. See [simulate_changes()].

`accepted_placements`, `link_origins`, `placement_weights`, `weight_col`
: Bundle scoring controls forwarded to the shared adapter, identical to [pagerank_screaming_frog()].

`apply_canonicals`, `apply_redirects`, `preset`
: Fold and view controls forwarded to the shared adapter, identical to [pagerank_screaming_frog()]. All of these shape the baseline and the proposed model equally, so the two differ only by the changeset.

`...`
: Additional scoring controls passed through to both [pagerank()] calls (e.g. ‘self_loops‘, ‘nofollow_action‘, ‘damping‘, ‘position_col‘).

`label_baseline`, `label_proposed`
: Labels for the two models in the comparison output. Defaults ‘"baseline"‘ / ‘"proposed"‘.

Value

The [simulate_changes()] output: a [compare_pagerank()] table with a ‘node_status‘ column, plus ‘summary‘, ‘proposed‘, and ‘manifest‘ attributes. See [simulate_changes()].

See Also

[simulate_changes()] for the bare edge-list entry point and full verb semantics; [pagerank_screaming_frog()] for scoring a bundle without a what-if.

Examples

```
internal <- data.frame(
  Address = c(
    "https://example.com/", "https://example.com/a", "https://example.com/b"
  ),
  `Status Code` = c("200", "200", "200"),
  check.names = FALSE
)
links <- data.frame(
  Type = c("Hyperlink", "Hyperlink"),
  Source = c("https://example.com/", "https://example.com/a"),
  Destination = c("https://example.com/a", "https://example.com/b"),
  Follow = c("TRUE", "TRUE"),
  check.names = FALSE
)
bundle <- screaming_frog_bundle(internal, links, "all_outlinks")

# Retire /a behind a redirect to /b
retire <- data.frame(
  from = "https://example.com/a", to = "https://example.com/b"
```

```
)
simulate_changes_screaming_frog(bundle, redirect_urls_df = retire)

# Model /a 404-ing (its inbound authority evaporates to the waste sink)
simulate_changes_screaming_frog(
  bundle,
  remove_urls = "https://example.com/a"
)
```

`smooth_transitions` *Structural Smoothing of Empirical Page Transitions*

Description

Shrinks sparse empirical transition shares toward the crawl-graph link structure, so that no valid crawled link is assigned a probability of exactly zero. Observed page-transition data (e.g. from [ga4_page_transitions()]) is sparse: a low-traffic but real internal link may simply never have been traversed in the measured window, which leaves its raw empirical share at zero and destabilises the stationary PageRank vector. This helper combines the empirical distribution with a structural prior using a per-source shrinkage weight that grows with the source page's sample size.

Usage

```
smooth_transitions(
  empirical_df,
  structural_df,
  k = 5,
  min_support = 0,
  lambda_fn = NULL,
  count_col = "n",
  structural_weight_col = NULL,
  from_col = "from",
  to_col = "to",
  prob_col = "transition_probability"
)
```

Arguments

`empirical_df`	A data frame of observed transitions: a 'from'/'to' edge list with a numeric count column. Typically the output of [ga4_page_transitions()] (or [aggregate_edges()] over behavioral counts). Duplicate 'from'/'to' rows are summed.
`structural_df`	A data frame of crawl-graph links forming the structural prior: a 'from'/'to' edge list, optionally with a numeric structural-weight column. Duplicate 'from'/'to' rows are summed (so repeated link instances raise the prior weight). When 'structural_weight_col' is 'NULL', each row contributes weight 1, i.e. a uniform prior over a source's crawled links.

`k` Positive numeric. The pseudocount / Dirichlet concentration controlling shrinkage strength in $\lambda_i = n_i/(n_i + k)$. Larger ‘k‘ pulls under-sampled sources more strongly toward the structural prior. Must be ‘> 0‘. Default ‘5‘. Ignored for a source if ‘lambda_fn‘ is given.

`min_support` Non-negative numeric. Sources with ‘0 < n_i < min_support‘ are treated as having insufficient empirical support and fall back fully to the structural prior ($\lambda_i = 0$). Default ‘0‘ (no minimum).

`lambda_fn` Optional function mapping a source’s sample size ‘n_i‘ (a single numeric) to a shrinkage weight in ‘[0, 1]‘. Overrides the default ‘n_i / (n_i + k)‘ rule for sources that have both empirical data and a structural prior and meet ‘min_support‘. Must return values ‘< 1‘ to preserve the non-zero-crawled-link guarantee. Default ‘NULL‘.

`count_col` Name of the numeric empirical-count column in ‘empirical_df‘. Default ‘"n"‘ (the [ga4_page_transitions()] default).

`structural_weight_col`
Name of an optional numeric structural-weight column in ‘structural_df‘, or ‘NULL‘ (default) for a uniform prior.

`from_col`, `to_col`
Names of the source / target columns, shared by both inputs and the output. Defaults ‘"from"‘ / ‘"to"‘.

`prob_col` Name of the output smoothed-probability column. Default ‘"transition_probability"‘. Pass this to ‘pagerank(weight_col = ...)‘.

Details

The shrinkage model

For each source page ‘i‘, the smoothed transition probability to target ‘j‘ is the convex combination

$$P(i \to j) = \lambda_i \cdot \mathrm{emp}(i \to j) + (1 - \lambda_i) \cdot \mathrm{prior}(i \to j)$$

where ‘emp(i -> j)‘ is the empirical share ‘count(i -> j) / n_i‘ (‘n_i‘ = total observed out-transitions from ‘i‘), ‘prior(i -> j)‘ is the structural prior share (the crawl-graph out-link distribution of ‘i‘), and λ_i is the per-source trust placed in the empirical data.

Sample-size-dependent shrinkage (‘lambda_i‘)

By default $\lambda_i = n_i/(n_i+k)$, the standard Dirichlet / pseudocount shrinkage rule: it is monotonically increasing in the source-page sample size ‘n_i‘ and equals ‘1/2‘ at ‘n_i = k‘. Equivalently, the model adds ‘k‘ pseudo-observations distributed according to the structural prior, then renormalizes — a Dirichlet prior with concentration ‘k‘. A high-traffic source (large ‘n_i‘) is trusted almost entirely to its own behavior; a barely-sampled source leans on the crawl structure. ‘k‘ must be strictly positive: this is precisely what guarantees $\lambda_i < 1$ for any sampled source, hence a non-zero $(1 - \lambda_i)$ weight on every crawled link (see *Guarantees*).

Per-source special cases

Sources are matched between the two inputs and resolved as follows: - **No empirical data** (‘n_i = 0‘; crawled link absent from behavioral data): $\lambda_i = 0$, so ‘P(i -> .)‘ is the pure structural prior. The crawl link still receives mass. - **No structural prior** (observed transition whose source has

no crawled out-links): $\lambda_i = 1$, so ‘P(i -> .)‘ is the pure empirical distribution — there is nothing to shrink toward. - **Insufficient support** (‘0 < n_i < min_support‘): $\lambda_i = 0$. The empirical sample is treated as too small to trust, and the source falls back to its structural prior (when one exists). - **Otherwise**: $\lambda_i = n_i/(n_i + k)$ (or ‘lambda_fn(n_i)‘).

Edge universe

The output covers the **union** of empirical and structural out-edges per source, with an ‘origin‘ column flagging each as ‘"both"‘, ‘"empirical_only"‘ (an observed transition absent from the crawl graph), or ‘"structural_only"‘ (a crawled link never observed behaviorally). Edges whose smoothed probability is exactly zero — only possible for an ‘empirical_only‘ edge from a below-‘min_support‘ source — are dropped, since they carry no transition mass.

Time decay and segmentation

Time decay and device / template / channel segmentation are handled **upstream** by shaping the count input, keeping this function focused on the shrinkage itself. For time decay, supply decayed (fractional) counts in ‘count_col‘ — the ‘n_i‘ totals and λ_i then reflect effective sample size, and ‘count_col‘ need not be integer. For segmentation, partition the empirical counts by segment and call ‘smooth_transitions()‘ once per segment (optionally against a segment-specific structural prior), then combine the results.

Guarantees

For any ‘k > 0‘, every crawled link present in ‘structural_df‘ receives a strictly positive smoothed probability: such an edge has ‘prior(i -> j) > 0‘, and either $\lambda_i = 0$ (pure prior) or $\lambda_i < 1$ (since ‘n_i / (n_i + k) < 1‘), so the $(1 - \lambda_i)$ weight on the prior is positive. Within each source, the returned probabilities sum to 1.

Value

A data frame with one row per surviving source-target edge (the per-source union of empirical and structural edges, zero-probability edges dropped), ordered by ‘from‘ then ‘to‘, carrying:

‘from_col‘, ‘to_col‘ the edge endpoints (character).

‘prob_col‘ the smoothed transition probability; sums to 1 within each source.

‘empirical_count‘ the observed count for this edge (0 if the edge is ‘structural_only‘).

‘empirical_share‘ ‘count / n_i‘, the raw empirical share (0 if the source had no empirical data for this edge).

‘structural_prior‘ the structural prior share for this edge (0 if absent from the crawl graph).

‘support‘ ‘n_i‘, the source page’s total empirical out-count.

‘lambda‘ the per-source shrinkage weight applied.

‘origin‘ ‘"both"‘, ‘"empirical_only"‘, or ‘"structural_only"‘.

See Also

[ga4_page_transitions()] for the empirical input, [aggregate_edges()] for collapsing behavioral counts, and [pagerank()] for consuming the smoothed probabilities via ‘weight_col = prob_col‘.

Examples

```
# Sparse behavioral data: only A->B was ever observed.
empirical <- data.frame(
  from = c("A", "A"),
  to = c("B", "C"),
  n = c(8, 0)
)[1, ]
# Crawl graph: A links to both B and C.
structural <- data.frame(
  from = c("A", "A"),
  to = c("B", "C")
)
smoothed <- smooth_transitions(empirical, structural, k = 5)
smoothed
# A->C keeps a non-zero probability despite never being observed.

# Feed to pagerank() as a smoothed behavioral transition model:
# pagerank(smoothed, weight_col = "transition_probability",
#          clean_edge_urls = FALSE)
```

---

summary.screaming_frog_bundle

Summarize a Screaming Frog bundle

Description

Summarize a Screaming Frog bundle

Usage

```
## S3 method for class 'screaming_frog_bundle'
summary(object, ...)
```

Arguments

object	A ‘screaming_frog_bundle‘ object.
...	Unused; for S3 compatibility.

Value

A compact named list of row counts and reconciliation counts.

Examples

```
internal <- data.frame(
  Address = c("https://example.com/", "https://example.com/a"),
  `Status Code` = c("200", "200"),
  check.names = FALSE
)
```

```
links <- data.frame(
  Type = "Hyperlink",
  Source = "https://example.com/",
  Destination = "https://example.com/a",
  Follow = "TRUE",
  check.names = FALSE
)
bundle <- screaming_frog_bundle(internal, links, "all_outlinks")
summary(bundle)
```

`topic_feeder_pagerank` *Topic Feeder PageRank (seeded reverse-graph PageRank)*

Description

The reverse-graph sibling of [topic_sensitive_pagerank()]. Where Topic-Sensitive PageRank answers *"given I care about this cluster, which pages are most **authoritative** for it?"* (authority flows downstream **from** the seed cluster), ‘topic_feeder_pagerank()‘ answers the inverse question: *"which pages **feed / power** this cluster?"* — i.e. the strongest internal hubs whose outlinks point **into** the target pages.

It is personalized PageRank with the teleport prior concentrated on the **target cluster**, run on the **transposed** link graph (‘reverse = TRUE‘). Mass teleports onto the cluster and then walks *backward* along links, so it accumulates on the pages that funnel authority toward the cluster. The further (in out-link hops) a page is from the cluster, the less feeder credit it earns — the PageRank damping factor is exactly that attenuation.

Like [trustrank()] and [topic_sensitive_pagerank()], this introduces **no new solver**: it builds a ‘prior_df‘ from the seed set ([seed_prior()]) and hands it to [pagerank()] with ‘reverse = TRUE‘. The caller supplies the cluster; there is no topic inference. The prior builder is the *same* [seed_prior()] that [trustrank()] uses — a seed prior is orientation- agnostic; the reversed graph is what makes this a feeder query.

Usage

```
topic_feeder_pagerank(
  edge_list_df,
  seeds,
  seed_weight = NULL,
  seed_url_col = "url",
  seed_weight_col = "weight",
  ...
)
```

Arguments

`edge_list_df`	A data frame representing the edge list, typically with columns like "from" and "to". Edges are expected to be page-to-page hyperlinks: ‘pagerank()‘ is graph-agnostic and treats every endpoint as a node, so resource links (images, CSS, JS,

and other non-HTML references) must be filtered upstream or they collect authority as ordinary vertices. ‘pagerank_screaming_frog()‘ does this at the crawl boundary via [sf_graph_eligible()] (‘Hyperlink‘ only); a hand-built or non-SF edge list should apply the same hyperlink-only filter before scoring.

`seeds` The target cluster. Either a character vector of cluster URLs (each gets equal seed weight unless ‘seed_weight‘ is given), or a data frame with a URL column and a numeric weight column (see ‘seed_url_col‘ / ‘seed_weight_col‘) for unequal cluster emphasis. See [seed_prior()].

`seed_weight` Optional numeric weight for a character-vector ‘seeds‘: either one value per seed or a single value recycled to all seeds. Ignored when ‘seeds‘ is a data frame. Default ‘NULL‘ (every cluster page weight ‘1‘, a uniform distribution over the cluster).

`seed_url_col, seed_weight_col`
Column names used when ‘seeds‘ is a data frame. Defaults ‘"url"‘ / ‘"weight"‘. Ignored for a character vector.

`...` Additional arguments forwarded to [pagerank()] (e.g. ‘redirects_df‘, ‘canonicals_df‘, ‘rurl_params‘, ‘weight_col‘, ‘prior_transform‘, ‘prior_alpha‘, ‘damping‘). Passing ‘prior_df‘, ‘prior_url_col‘, ‘prior_weight_col‘, or ‘reverse‘ is an error.

Details

Why this is not recoverable from forward PageRank

In PageRank, authority flows **along** link direction: linking *to* an important page does not make the linker important. So "the pages that feed cluster X" is **not** a re-reading of forward (or Topic-Sensitive) PageRank scores — it is the reversed-graph notion. Forward PageRank and [topic_sensitive_pagerank()] rank pages by **inflow** (important because important pages point at them); ‘topic_feeder_pagerank()‘ ranks by **outflow toward the cluster** (important because *it* points at the cluster).

How to read the result

The seed (cluster) pages carry the teleport mass directly, so they appear in ‘prior_weight‘ with a positive value and tend to score highly *by construction* — that is teleport, not a feeder signal. **The feeders are the high-‘pagerank‘ pages whose ‘prior_weight‘ is ‘0‘** (pages outside the cluster that nonetheless accumulate reverse-walk mass). Rank by ‘pagerank‘ and read off the top non-seed rows, or filter ‘prior_weight == 0‘.

Relationship to neighboring tools

[pagerank() (forward)] Global inflow authority. Feeder PageRank is its transpose, biased to a cluster.

[topic_sensitive_pagerank() (G2)] The forward-graph sibling: same personalization plumbing, opposite flow direction. G2 finds a cluster's *authorities*; this finds its *feeders*. They are complementary, not substitutes.

Inverse PageRank (‘pagerank(reverse = TRUE)‘) The **global**, unseeded outflow centrality — "which pages funnel authority outward anywhere on the site". Feeder PageRank adds the cluster bias: not "good hub in general" but "good hub *for this cluster*". With no ‘seeds‘ you would just call ‘pagerank(reverse = TRUE)‘ directly.

HITS hubs ([hits()]) Also an outflow notion, but a co-computed eigenvector pair (hub <-> authority) with no teleport prior and no damped-surfer / dangling handling. Feeder PageRank is the random-surfer-model, cluster-seedable counterpart that stays inside the [pagerank()] graph-preparation contract (redirects, canonicals, duplicate-edge policy, weights).

Seed weights are an **additive feeder budget**: if two seed URLs fold onto the same vertex (redirect / canonical variants) their weights sum, exactly as the [pagerank()] / [align_prior_to_vertices()] prior contract specifies.

Everything [pagerank()] accepts flows through '...': redirects, canonicals, URL cleaning, domain/host filtering, edge weights, duplicate-edge policy, and the prior-shaping knobs ('prior_transform', 'prior_alpha'). Because this owns both the prior and the graph orientation, passing 'prior_df', 'prior_url_col', 'prior_weight_col', or 'reverse' is an error. Note that the direction-sensitive forward-flow devices that [pagerank()] already rejects under 'reverse = TRUE' ('nofollow_action = "evaporate"', 'indexability_df') are likewise unavailable here.

Value

The [pagerank()] result data frame ('node_name', 'pagerank', and the 'prior_weight' column the prior path adds), sorted by 'pagerank' descending, carrying the usual '"transition_audit"' attribute. The audit's model configuration records 'reverse = TRUE'.

See Also

[seed_prior()], [topic_sensitive_pagerank()], [trustrank()], [pagerank()], [align_prior_to_vertices()], [hits()]

Examples

```
edges <- data.frame(
  from = c("/hub", "/hub", "/feeder", "/blog", "/ai", "/news"),
  to = c("/ai", "/ai-demo", "/ai", "/ai", "/ai-demo", "/sports")
)

# Which pages feed the AI-Agent cluster?
fr <- topic_feeder_pagerank(
  edges,
  seeds = c("/ai", "/ai-demo"),
  clean_edge_urls = FALSE
)

# Feeders are the top-scoring rows OUTSIDE the cluster (prior_weight == 0).
fr[fr$prior_weight == 0, c("node_name", "pagerank")]

# Build the cluster prior explicitly and run it yourself, if you prefer:
prior <- seed_prior(c("/ai", "/ai-demo"))
identical_run <- pagerank(
  edges, prior_df = prior, reverse = TRUE, clean_edge_urls = FALSE
)
```

`topic_sensitive_pagerank` *Topic-Sensitive PageRank (multi-vector personalized PageRank)*

Description

Computes a per-topic PageRank by running the standard [pagerank()] engine once per topic, biasing the random surfer's teleport toward each topic's seed pages, then optionally blends the per-topic scores into a single combined ranking.

This is Haveliwala's (2002) Topic-Sensitive PageRank adapted to a single site: instead of one global ranking, each "topic" is a content cluster (e.g. the *pricing* section, the *AI-Agent* product area, the *support* docs) defined by a set of seed URLs. A page can be highly authoritative for one topic and unimportant for another on the *same* link graph — the only thing that changes between runs is where the surfer teleports.

Mechanically this is pure orchestration over the existing TIPR personalization path: each topic becomes a ‘prior_df‘ handed to [pagerank()] (see [align_prior_to_vertices()]). There is no new solver and no topic inference — the caller supplies the seed sets.

Usage

```
topic_sensitive_pagerank(
  edge_list_df,
  topics,
  topic_weights = NULL,
  topic_url_col = "url",
  topic_weight_col = "weight",
  ...
)
```

Arguments

`edge_list_df`	A data frame representing the edge list, typically with columns like "from" and "to". Edges are expected to be page-to-page hyperlinks: ‘pagerank()‘ is graph-agnostic and treats every endpoint as a node, so resource links (images, CSS, JS, and other non-HTML references) must be filtered upstream or they collect authority as ordinary vertices. ‘pagerank_screaming_frog()‘ does this at the crawl boundary via [sf_graph_eligible()] (‘Hyperlink‘ only); a hand-built or non-SF edge list should apply the same hyperlink-only filter before scoring.
`topics`	A **uniquely named** list, one element per topic. Each element defines that topic's teleport seed set and is either: **a character vector of seed URLs** each seed gets equal weight ‘1‘; **a data frame** with a URL column and a numeric weight column (see ‘topic_url_col‘ / ‘topic_weight_col‘) for weighted seeds. The list names become the per-topic score column names in the result, so they must be non-empty, unique, and must not be the reserved names ‘"node_name"‘

or ‘"blended"‘. Seed URLs are canonicalized and redirect- / canonical-folded into the graph’s vertex namespace by [pagerank()] before alignment, identically to any other prior.

`topic_weights` Optional blend weights for the ‘blended‘ column. Either a named numeric whose names match ‘topics‘, or an unnamed numeric of the same length as ‘topics‘ (applied in list order). Must be non-negative, finite, and sum to a positive value; they are normalized to sum to 1 internally. Default ‘NULL‘ gives every topic equal weight.

`topic_url_col, topic_weight_col`
Column names used when a topic is supplied as a data frame. Defaults ‘"url"‘ / ‘"weight"‘. Ignored for topics given as plain character vectors.

`...` Additional arguments forwarded to [pagerank()] and onward to ‘igraph::page_rank‘ (e.g. ‘redirects_df‘, ‘canonicals_df‘, ‘rurl_params‘, ‘weight_col‘, ‘prior_transform‘, ‘prior_alpha‘, ‘damping‘). Because this function owns the teleport prior, passing ‘prior_df‘, ‘prior_url_col‘, or ‘prior_weight_col‘ here is an error — supply ‘topics‘ instead. Inner per-topic alignment diagnostics are silenced by default (‘prior_verbose = FALSE‘); pass ‘prior_verbose = TRUE‘ to re-enable them.

Details

All topics are scored on the **same** prepared graph: graph construction (URL cleaning, redirect/canonical folding, domain/host filtering, duplicate and isolate handling) depends only on ‘edge_list_df‘ and the forwarded options, never on the teleport prior, so the vertex set is identical across topics. The per-topic results are combined with a full outer join on ‘node_name‘; any node missing from a topic (which can only happen if you opt into ‘prior_inject_unmatched = TRUE‘, where unmatched seed URLs are injected as topic-specific isolates) is filled with score ‘0‘ for that topic.

The ‘blended‘ column is the weight-normalized linear combination $\sum_t w_t \cdot score_t$. Each per-topic column carries the same mass semantics as a single [pagerank()] run (it can sum to less than 1 under nofollow evaporation or ‘robots_blocked_action = "vanish"‘), and the blend inherits that — it is a weighted average of the per-topic distributions, not renormalized.

Value

A data frame with one row per node, sorted by ‘blended‘ descending:

node_name Node identifier (shared vertex namespace).

<one column per topic> The personalized PageRank score for that topic, named after the corresponding ‘topics‘ entry.

blended The ‘topic_weights‘-weighted combination of the per-topic scores.

Two attributes are attached: ‘"topic_weights"‘, the normalized weights used for the blend, and ‘"topic_audits"‘, a named list of the per-topic [transition_audit] objects from each underlying [pagerank()] run.

See Also

[pagerank()], [align_prior_to_vertices()], [compare_pagerank()]

Examples

```
edges <- data.frame(
  from = c("/", "/", "/", "/ai", "/ai", "/blog", "/pricing"),
  to = c("/ai", "/blog", "/pricing", "/ai-demo", "/pricing", "/ai", "/")
)

# Two topics: the AI cluster and the pricing cluster.
res <- topic_sensitive_pagerank(
  edges,
  topics = list(
    ai_agent = c("/ai", "/ai-demo"),
    pricing = "/pricing"
  ),
  clean_edge_urls = FALSE
)
print(res)

# Bias the blend 70/30 toward the AI cluster.
res2 <- topic_sensitive_pagerank(
  edges,
  topics = list(
    ai_agent = c("/ai", "/ai-demo"),
    pricing = "/pricing"
  ),
  topic_weights = c(ai_agent = 0.7, pricing = 0.3),
  clean_edge_urls = FALSE
)
attr(res2, "topic_weights")
```

`transform_edge_weights` *Transform Edge Weights Per Source (Grouped)*

Description

Applies a weight transformation *within each source page's outgoing choice set*, rather than across one global vector. Link ranks and transition weights are normally meaningful relative to the other links on the *same* source page: a "position 1" link on page A and a "position 1" link on page B should each be top-of-choice-set for their own source. A global rank (as computed by `transform_weights`) conflates them; this helper computes the transform separately within each by group.

In addition to the transformed weight, it returns a normalized `transition_probability` that sums to 1 within each by group, so the per-source choice distribution can be inspected and validated before it reaches the solver (igraph re-normalizes edge strengths internally, but that normalization is not otherwise visible to the user).

Usage

```
transform_edge_weights(
```

```
  edge_list_df,
  value_col,
  by = "from",
  method = "zipf",
  weight_col = "weight",
  prob_col = "transition_probability",
  ...
)
```

Arguments

edge_list_df	A data frame of edges. Must contain the column named by by and the column named by value_col.
value_col	Character, the name of the column holding the raw numeric signal to transform (e.g. link positions, GA4 click counts).
by	Character, the name of the grouping column defining each choice set. Default "from" (the source page). May name multiple columns to group by their combination.
method	Character, the transformation strategy, passed through to transform_weights. One of "none", "log", "percentile", "minmax", "zipf", "rank_linear". Default "zipf".
weight_col	Character, the name of the output column to hold the transformed weight. Default "weight".
prob_col	Character, the name of the output column to hold the per-source normalized transition_probability. Default "transition_probability".
...	Additional arguments forwarded to transform_weights (e.g. alpha, offset, floor_value, descending).

Details

The transform is applied independently per group by calling transform_weights on each group's slice of value_col – it reuses, rather than re-implements, the existing methods. transition_probability is then formed by dividing each group's transformed weights by their group sum. NA transformed weights (e.g. from NA inputs) are carried through and excluded from the probability total. A group whose transformed weights sum to zero (or are all NA) yields NA probabilities for that group, since no meaningful distribution can be formed.

Value

The input data frame with two columns added (or overwritten): weight_col (the per-source transformed weight) and prob_col (the per-source transition probability, summing to 1 within each by group across non-NA weights). Row order is preserved.

See Also

transform_weights for the single-vector (global) transform and the full description of each method.

Examples

```
# Two source pages, each with its own link positions (1 = top)
edges <- data.frame(
  from = c("A", "A", "A", "B", "B"),
  to = c("B", "C", "D", "C", "D"),
  position = c(1, 2, 3, 1, 2)
)

# Zipf weights computed within each source's choice set
transform_edge_weights(edges, "position",
  method = "zipf", descending = FALSE
)

# The transition_probability column sums to 1 within each `from`
```

`transform_weights` *Transform Edge Weights for PageRank*

Description

Applies a transformation strategy to a numeric vector of edge weights before passing them to `pagerank`. Useful for converting link positions, click counts, or other raw signals into weights suitable for the PageRank random surfer model.

Usage

```
transform_weights(
  x,
  method = c("none", "log", "percentile", "minmax", "zipf", "rank_linear"),
  alpha = 1,
  offset = 1,
  floor_value = 0.01,
  descending = TRUE
)
```

Arguments

`x`	Numeric vector of raw weights (e.g., link positions on a page, GA4 click counts, or any positive numeric signal).
`method`	Character, the transformation strategy. One of:
	`"none"` Return `x` unchanged.
	`"log"` Apply `log(x + offset)` to compress large ranges (e.g., GA4 click counts spanning 1 to 100,000). The `offset` parameter (default 1) avoids `log(0)`.
	`"percentile"` Map values to their empirical percentile (0–1). Robust to extreme outliers.

"minmax" Scale to the [0, 1] range using min-max normalisation. A small floor (floor_value, default 0.01) is added so that the lowest-weighted edge still carries some weight rather than zero.

"zipf" Convert to rank order, then apply Zipf's law: weight = 1 / rank^alpha. Position 1 gets 1.0, position 2 gets 1/2^alpha, etc. Controlled by the alpha parameter (default 1).

"rank_linear" Convert to rank order (1 = highest value), then assign linearly decreasing weights: weight = (n - rank + 1) / n. Position 1 gets 1.0, position n gets 1/n.

alpha — Numeric, exponent for the "zipf" method. Default 1.0. Higher values make the drop-off steeper (position 1 dominates more).

offset — Numeric, added to x before the "log" transform. Default 1 (so that zero-valued inputs produce log(1) = 0 rather than -Inf).

floor_value — Numeric, minimum weight for the "minmax" method. Default 0.01.

descending — Logical. For rank-based methods ("rank_linear", "zipf"), whether higher input values get higher weights. Default TRUE (e.g., if the input is click counts, more clicks = higher weight). Set to FALSE when the input is link position on a page (position 1 = most valuable, but numerically smallest).

Value

Numeric vector of the same length as x with transformed weights. NA values in x are preserved as NA in the output.

Examples

```
# Link positions on a page (1 = top, most valuable)
positions <- c(1, 2, 3, 4, 5)
transform_weights(positions, "rank_linear", descending = FALSE)
transform_weights(positions, "zipf", alpha = 1, descending = FALSE)
transform_weights(positions, "zipf", alpha = 2, descending = FALSE)

# GA4 click counts (wide range)
clicks <- c(50000, 12000, 800, 150, 3)
transform_weights(clicks, "log")
transform_weights(clicks, "minmax")
transform_weights(clicks, "zipf")

# Use with pagerank()
edges <- data.frame(
  from = c("Home", "Home", "Home"),
  to = c("About", "Blog", "Contact"),
  position = c(1, 2, 5)
)
edges$weight <- transform_weights(edges$position, "zipf",
  descending = FALSE
)
# pagerank(edges, weight_col = "weight", clean_edge_urls = FALSE)
```

`transition_audit` *Transition-construction audit / provenance object*

Description

Builds a stable, documented audit / provenance record describing what happened to the edges and weights as [pagerank()] turned a raw edge list into the transition graph it scored. It is the backbone of reproducibility and of downstream diagnostics: it carries the row/edge counts, behavioral-weight coverage, normalization totals, the data that was dropped along the way (rows lost to NA / deduplication / self-loop removal, and authority-prior URLs that never folded onto a vertex), and the relevant [pagerank()] configuration. It also records the duplicate-edge policy used to build transitions, so callers can distinguish the default destination-level surfer from opt-in aggregate / link-slot models.

Details

Structure and contract

The object is an S3 list with class ‘"transition_audit"‘ (a list was chosen over a bare list so that it prints a human-readable summary while remaining a plain, inspectable ‘list‘ for programmatic access — ‘audit$counts$n_edges‘ works as expected, mirroring the existing [audit_redirects()] / [audit_canonicals()] objects in this package). The documented top-level fields are **stable**; callers may rely on them being present.

counts A list of integer counts: ‘n_input_rows‘ (rows in the raw ‘edge_list_df‘), ‘n_edges‘ (directed edges remaining after URL folding, deduplication and self-loop handling — i.e. the edges actually scored), and ‘n_vertices‘ (vertices in the returned result).

coverage A list describing behavioral-weight coverage: ‘weighted‘ (logical, whether a ‘weight_col‘ was in effect), ‘weight_col‘ (its name or ‘NULL‘), ‘n_edges_weighted‘ (edges carrying a finite, positive weight), and ‘coverage‘ (the fraction ‘n_edges_weighted / n_edges‘, or ‘NA_real_‘ when there are no edges / no weighting).

normalization A list of normalization totals: ‘pagerank_total‘ (sum of the returned PageRank scores; ‘< 1‘ when mass evaporated via nofollow, vanished robots-blocked pages, etc.).

dropped A list accounting for data removed during construction: ‘n_rows_na‘ (input rows dropped because ‘from‘/‘to‘ was ‘NA‘), ‘n_rows_duplicate‘ (rows collapsed by edge deduplication), ‘n_self_loops‘ (self-loop edges dropped when ‘self_loops = "drop"‘), ‘n_rows_collapsed‘ (total input rows that did not survive as distinct scored edges = ‘n_input_rows - n_edges‘), ‘n_prior_unmatched‘ (authority prior URLs that did not fold onto any vertex; ‘NA_integer_‘ when no ‘prior_df‘ was supplied), ‘n_robots_blocked‘ (URLs treated as robots.txt-blocked), and ‘n_status_dead‘ (in-graph URLs whose HTTP status code marked them response-dead; ‘0‘ when no ‘status_df‘ was supplied).

duplicates A list describing duplicate-edge handling: ‘policy‘ (the ‘duplicate_edge_policy‘ passed to [pagerank()]), ‘n_duplicate_rows‘ (post-fold duplicate input rows), ‘instance_count_col‘ (the internal audit column used by ‘"count_instances"‘, or ‘NULL‘), and ‘n_duplicate_instances‘ (the number of duplicate link instances folded into transition weights), and ‘duplicate_edges‘ (a compact data frame of counted edges with more than one link instance, or ‘NULL‘).

config A list of the [pagerank()] arguments that materially shape the transition graph. ‘preset‘ records the *provenance* of the rest: the name of the [pr_preset()] bundle the caller asked for (e.g. ‘"declared"‘), ‘"custom"‘ for a hand-rolled bundle, or ‘NULL‘ when no preset was used — so a run made as a named view stays distinguishable from the same arguments typed out by hand. ‘placement‘ records placement-aware weighting when it was used (‘placement_col‘, ‘accepted_placements‘, ‘placement_weights‘, and ‘n_rows_dropped‘, the number of edge rows the placement filter removed), or ‘NULL‘ when it was not — so a downweighted edge can be explained by the region it sits in rather than only by the opaque weight column it produced. ‘boilerplate‘ records the recurrence detector the same way when it was used (‘container_col‘, ‘boilerplate_threshold‘, ‘min_container_pages‘, ‘boilerplate_weight‘, and the counts ‘n_containers‘, ‘n_edges_scored‘, ‘n_edges_judged‘ and ‘n_edges_discounted‘), or ‘NULL‘ when it was not. Placement and recurrence are two detectors feeding one graded axis, and the strongest applicable discount wins, so both are recorded **separately**: the resulting weight alone cannot say which detector produced it. ‘position‘ records the orthogonal reading-order axis the same way when it was used (‘position_col‘, ‘position_transform‘, ‘position_alpha‘, ‘position_floor‘, and the counts ‘n_edges_scored‘, ‘n_sources_scored‘, and ‘min_position_weight‘), or ‘NULL‘ when it was not – it *multiplies* into the weight rather than competing for the minimum, so it too is recorded on its own so an edge weighing ‘0.02‘ can be explained as region times reading order. The other fields are the resolved configuration itself: ‘self_loops‘, ‘drop_isolates_flag‘, ‘reverse‘, ‘weight_col‘, ‘nofollow_col‘, ‘nofollow_action‘, ‘robots_blocked_action‘, ‘prior_alpha‘, ‘prior_transform‘, ‘prior_inject_unmatched‘, and the logical flags ‘has_redirects‘ / ‘has_canonicals‘ (whether that signal *materially* folded an edge — an effective no-op such as a self-canonical reads ‘FALSE‘), ‘has_indexability‘, and ‘has_prior‘.

mass A list decomposing the internal stationary vector (which always sums to 1) into its accounted-for components: ‘reported‘ (the mass on returned, visible pages — equals the summed result scores), ‘sink‘ (the **evaporated mass**: authority routed to the shared waste sink — what the whole waste class (noindex / robots-blocked / 4xx-5xx) and every real nofollowed link under ‘nofollow_action = "evaporate"‘ passed on but could not deliver), ‘leaked‘ (the **leaked mass**: authority sent to the synthetic leak sink under ‘out_of_scope_fold = "leak"‘, i.e. equity that flowed into out-of-scope-folded sources and left the measured graph — ‘0‘ when no leak occurred), ‘hidden‘ (the **hidden mass**: the own stationary mass of robots-blocked nodes removed under ‘robots_blocked_action = "vanish"‘; their pass-through still routes to the waste sink and is counted in ‘sink‘), and ‘total‘ (their sum, which reconciles to 1 by construction). These are the precise components of the deficit between the reported scores and 1 — it is evaporated, leaked and hidden mass, not undifferentiated "leakage". Each is ‘NULL‘ when the stationary vector is undefined (e.g. an empty graph).

fold A list recording how **out-of-scope folds** were handled — a composed fold-map entry whose *target* (the representative a crawled source folds onto) is not itself a crawled node, which silently invents a phantom vertex. ‘policy‘ (the ‘out_of_scope_fold‘ argument, ‘"relabel"‘, ‘"keep"‘ or ‘"leak"‘), ‘n_out_of_scope‘ (count of such entries), ‘applied‘ (logical: ‘TRUE‘ when they were acted upon — relabeled / folded through under ‘"relabel"‘, or routed to the leak sink under ‘"leak"‘ — and ‘FALSE‘ when skipped / kept as crawled under ‘"keep"‘; combine with ‘policy‘ to distinguish relabel from leak), and ‘out_of_scope‘ (a data frame of the offending ‘source‘ / ‘target‘ / ‘signal‘ rows, or ‘NULL‘ when there were none), and ‘collisions‘ (a data frame of **fold-target collisions** — uncrawled URLs that a fold relabeled a crawled source onto while they were ALSO independently linked, so the two silently merge into one vertex and the crawled page absorbs the inbound link equity of that uncrawled URL;

columns ‘target‘, ‘n_independent_refs‘ and the folded ‘source‘(s) — or ‘NULL‘ when none). A collision triggers a ‘warning()‘ naming the merged URL(s). This diagnostic requires crawl-URL knowledge to distinguish an uncrawled fold target from a genuinely crawled leaf page, so it is only computed when an ‘indexability_df‘ is supplied to [pagerank()]; without it, ‘collisions‘ is ‘NULL‘. Recorded regardless of ‘out_of_scope_fold‘ policy.

The constructor [new_transition_audit()] is internal plumbing for [pagerank()]; the object is normally obtained via ‘attr(result, "transition_audit")‘ (see [pagerank()]).

See Also

[pagerank()], [audit_redirects()], [audit_canonicals()]

Examples

```
# A transition_audit is attached to every pagerank() result, and explains
# what happened between the raw edge list and the graph actually scored.
edges <- data.frame(
  from = c("/a", "/a", "/b", "/b", "/c", NA),
  to = c("/b", "/b", "/c", "/b", "/a", "/a")
)
result <- pagerank(edges, self_loops = "drop")
audit <- attr(result, "transition_audit")
audit

# The documented top-level fields are stable, so callers can rely on them.
audit$counts$n_input_rows # 6 raw rows in ...
audit$counts$n_edges # ... 3 distinct edges scored

# Each collapsed row is accounted for individually.
audit$dropped$n_rows_na # the NA-endpoint row
audit$dropped$n_rows_duplicate # the repeated /a -> /b row
audit$dropped$n_self_loops # the /b -> /b self-loop

# Mass accounting: the internal stationary vector always sums to 1, split
# into reported (visible) mass plus whatever evaporated / leaked / hid.
audit$mass$reported
audit$mass$total
```

---

trustrank *TrustRank Seed-Biased PageRank*

Description

TrustRank (Gyöngyi, Garcia-Molina & Pedersen, 2004) is personalized PageRank whose teleport vector is concentrated on a set of **trusted seed** pages instead of being uniform. Trust then flows outward along links and attenuates with distance (the PageRank damping factor *is* the trust-attenuation mechanism), so pages well-linked from the trusted core score high and pages far from it score low.

‘pagerankr‘ implements this with **no new solver**: a trusted-seed prior is exactly a ‘prior_df‘ for the existing TIPR personalization path. Build that prior from a seed set with [seed_prior()], and ‘trustrank()‘ is the worked convenience wrapper that builds the seed prior and runs [pagerank()] with it on the forward graph.

This is **seed-biased PageRank**, not a full spam-detection system: it reproduces the biased-propagation core of TrustRank, leaving seed selection (expert-reviewed "good" pages) to the caller.

Usage

```
trustrank(
  edge_list_df,
  seeds,
  seed_weight = NULL,
  seed_url_col = "url",
  seed_weight_col = "weight",
  ...
)
```

Arguments

`edge_list_df`	A data frame representing the edge list, typically with columns like "from" and "to". Edges are expected to be page-to-page hyperlinks: ‘pagerank()‘ is graph-agnostic and treats every endpoint as a node, so resource links (images, CSS, JS, and other non-HTML references) must be filtered upstream or they collect authority as ordinary vertices. ‘pagerank_screaming_frog()‘ does this at the crawl boundary via [sf_graph_eligible()] (‘Hyperlink‘ only); a hand-built or non-SF edge list should apply the same hyperlink-only filter before scoring.
`seeds`	The trusted seed set. Either a character vector of trusted URLs (each gets equal seed weight unless ‘seed_weight‘ is given), or a data frame with a URL column and a numeric weight column (see ‘seed_url_col‘ / ‘seed_weight_col‘) for unequal trust. See [seed_prior()].
`seed_weight`	Optional numeric trust weight for a character-vector ‘seeds‘: either one value per seed or a single value recycled to all seeds. Ignored when ‘seeds‘ is a data frame. Default ‘NULL‘ (every seed weight ‘1‘, i.e. a uniform distribution over the trusted set, as in the original TrustRank).
`seed_url_col, seed_weight_col`	Column names used when ‘seeds‘ is a data frame. Defaults ‘"url"‘ / ‘"weight"‘. Ignored for a character vector.
`...`	Additional arguments forwarded to [pagerank()] (e.g. ‘redirects_df‘, ‘rurl_params‘, ‘prior_transform‘, ‘prior_alpha‘, ‘damping‘). Passing ‘prior_df‘, ‘prior_url_col‘, or ‘prior_weight_col‘ is an error.

Details

The seed weights are an **additive trust budget**: when two seed URLs fold onto the same vertex (redirect/canonical variants) their weights sum, exactly as the [pagerank()] / [align_prior_to_vertices()] prior contract specifies. Equal weights reproduce TrustRank’s uniform seed distribution; unequal

weights express graded trust. See [seed_prior()] for the prior-builder contract; the same builder serves [topic_feeder_pagerank()], which runs it on the reversed graph.

‘trustrank()‘ forwards ‘...‘ to [pagerank()], so the full graph-preparation surface (redirects, canonicals, URL cleaning, domain/host filtering, edge weights, duplicate-edge policy) and the prior-shaping knobs (‘prior_transform‘, ‘prior_alpha‘) are all available. In particular ‘prior_alpha‘ mixes a uniform teleport baseline back in: ‘prior_alpha = 0‘ (the default) is pure trust teleport (untrusted, unreachable pages get no teleport mass), while a small positive value gives every page a floor. Because this owns the prior, passing ‘prior_df‘, ‘prior_url_col‘, or ‘prior_weight_col‘ to ‘trustrank()‘ is an error — supply ‘seeds‘.

Value

The [pagerank()] result data frame (‘node_name‘, ‘pagerank‘, and the ‘prior_weight‘ column the prior path adds), carrying the usual ‘"transition_audit"‘ attribute.

See Also

[seed_prior()], [pagerank()], [align_prior_to_vertices()], [topic_sensitive_pagerank()], [topic_feeder_pagerank()]

Examples

```
edges <- data.frame(
  from = c("/", "/", "/hub", "/hub", "/spam", "/good"),
  to = c("/hub", "/good", "/good", "/deep", "/good", "/hub")
)

# Build a trusted-seed prior, then run it through pagerank() manually.
prior <- seed_prior(c("/", "/hub"))
pr <- pagerank(edges, prior_df = prior, clean_edge_urls = FALSE)

# ...or in one call with the convenience wrapper.
tr <- trustrank(edges, c("/", "/hub"), clean_edge_urls = FALSE)
print(tr)
```

---

validate_edge_weights *Validate Edge Weights and Per-Source Totals*

Description

Inspect a weighted edge list before it reaches the PageRank solver. The report identifies negative and non-finite weights, degenerate sources whose outgoing weights are all zero, and (optionally) source totals that do not match an expected probability total.

Usage

```
validate_edge_weights(
  edge_list_df,
  weight_col = "weight",
  from_col = "from",
  expected_total = NULL,
  tolerance = sqrt(.Machine$double.eps),
  action = c("error", "warning", "none")
)
```

Arguments

edge_list_df	A data frame containing source and weight columns.
weight_col	Name of the numeric edge-weight column.
from_col	Name of the source-node column used to define outgoing choice sets.
expected_total	Optional finite, non-negative total expected for each source. Use ‘1‘ to validate an already-normalized transition-probability column. ‘NULL‘ (default) reports totals without enforcing a target.
tolerance	Non-negative numeric tolerance for ‘expected_total‘.
action	How validation failures are handled: ‘"error"‘ (default), ‘"warning"‘, or ‘"none"‘. The report is returned in every mode.

Value

A data frame with one row per source and columns describing edge count, weight total, invalid-value counts, all-zero status, optional total agreement, and overall validity.

Examples

```
edges <- data.frame(
  from = c("A", "A", "B"),
  to = c("B", "C", "C"),
  probability = c(0.25, 0.75, 1)
)
validate_edge_weights(
  edges,
  weight_col = "probability",
  expected_total = 1
)
```

Index