

Package ‘cayleyR’

July 29, 2026

Type Package

Title Cayley Graph Analysis for Permutation Puzzles

Version 0.2.6

Description Implements algorithms for analyzing Cayley graphs of permutation groups for the TopSpin puzzle. Provides methods for cycle detection, state space exploration and finding optimal operation sequences in permutation groups generated by shift and reverse operations. Also provides rule-defined landmark states for probing graphs too large to enumerate, and convex and non-convex hulls for measuring the solid such states span. The method Iterative Cycle Intersection (ICI) is described in Yuri Baramykov (2026) [doi:10.48550/arXiv.2607.13219](https://doi.org/10.48550/arXiv.2607.13219).

License MIT + file LICENSE

Encoding UTF-8

RoxygenNote 7.3.3

Imports Rcpp

LinkingTo Rcpp

Suggests testthat (>= 3.0.0), ggmlR, data.table, knitr, rmarkdown

Config/testthat/edition 3

VignetteBuilder knitr

URL <https://github.com/Zabis13/cayleyR>,
<https://arxiv.org/abs/2607.13219>

BugReports <https://github.com/Zabis13/cayleyR/issues>

NeedsCompilation yes

Author Yuri Baramykov [aut, cre] (ORCID:
<https://orcid.org/0009-0000-7627-4217>)

Maintainer Yuri Baramykov <lbsbmsu@mail.ru>

Repository CRAN

Date/Publication 2026-07-29 21:50:07 UTC

Contents

analyze_top_combinations	3
apply_operations	4
apply_operations_batch_gpu	5
bidirectional_bfs	6
breakpoint_distance	6
build_permutation_Matrix	7
calculate_angular_distance_z	8
calculate_differences	8
calculate_midpoint_z	9
cayley_bfs_full	10
cayley_gpu_available	11
cayley_gpu_free	11
cayley_gpu_init	12
cayley_gpu_status	12
cayley_graph_diameter	13
check_duplicates	14
compose_permutation_Matrix	15
convert_digits	15
convert_LRX_to_celestial	16
convex_hull_3d	17
create_state_store	18
cycle_shortcut	18
distance_methods	20
enclosing_hull_3d	21
find_best_match_human	22
find_best_random_combinations	23
find_closest_to_coords	25
find_combination_in_states	25
find_path_bfs	26
find_path_iterative	27
generate_state	29
generate_unique_states_df	30
get_reachable_states	30
get_reachable_states_light	31
human_algorithm	32
human_algorithm_to	33
human_phase1_navigate	34
human_phase1_rank	35
invert_path	36
landmark_states	37
manhattan_distance	38
manhattan_distance_Matrix_gpu	39
openmp_threads	40
reconstruct_bfs_path	40
reverse_prefix	41
reverse_prefix_simple	41

<i>analyze_top_combinations</i>	3
---------------------------------	---

run_length	42
save_bridge_states	43
select_unique	43
shift_left	44
shift_left_simple	45
shift_right	45
shift_right_simple	46
short_path_bfs	46
short_position	47
sparse_bfs	48
state_store_query	48
store_add_from_df	49
store_analyze_combos	50
store_analyze_combos_gpu	50
store_clear	51
store_clear_opd	51
store_collect_ops	52
store_combos_for_state	52
store_filter_middle	53
store_find_best_match	53
store_find_intersections	54
store_get_meta	54
store_get_state	55
store_lookup	55
store_reconstruct_path	56
store_set_opd	56
store_to_dataframe	57
validate_and_simplify_path	57
Index	59

analyze_top_combinations	
<i>Analyze Top Operation Combinations</i>	

Description

For each combination in a data frame of top results, runs a full cycle analysis and collects all states with their celestial coordinates into a single data frame.

Usage

analyze_top_combinations(top_combos, start_state, k)

Arguments

top_combos	Data frame or data.table with a combination column (string of operation digits, e.g., "132")
start_state	Integer vector, the initial permutation state
k	Integer, parameter for reverse operations

Value

Data frame with columns V1..Vn, operation, step, combo_number, nL, nR, nX, theta, phi, omega_conformal

Examples

```
combos <- data.frame(combination = c("13", "23"), stringsAsFactors = FALSE)
# result <- analyze_top_combinations(combos, 1:10, k = 4)
```

apply_operations	<i>Apply Sequence of Operations</i>
------------------	-------------------------------------

Description

Applies a sequence of shift and reverse operations to a permutation state. Operations can be specified as "1"/"L" (shift left), "2"/"R" (shift right), or "3"/"X" (reverse prefix). Tracks celestial coordinates.

Usage

```
apply_operations(state, operations, k, coords = NULL, compute_coords = TRUE)
```

Arguments

state	Integer vector representing the current permutation state
operations	Character vector of operations ("1"/"L", "2"/"R", "3"/"X")
k	Integer, parameter for reverse operations
coords	Optional list of current celestial coordinates. If NULL, starts from zero coordinates.
compute_coords	Logical, whether to track celestial coordinates. If FALSE, the returned coords component is NULL.

Value

List with components:

state	Integer vector after all operations applied
coords	List of final celestial coordinates (nL, nR, nX, theta, phi, omega_conformal)

Examples

```
result <- apply_operations(1:10, c("1", "3", "2"), k = 4)
result$state

# Using letter codes
result <- apply_operations(1:20, c("L", "X", "R"), k = 4)
result$state
```

apply_operations_batch_gpu

Apply operations to batch of states on GPU

Description

Applies a sequence of permutation operations to multiple states simultaneously using matrix multiplication on the Vulkan backend.

Usage

```
apply_operations_batch_gpu(states_matrix, operations, k)
```

Arguments

states_matrix	Numeric matrix (nrow x n), each row is a state
operations	Character vector of operation codes (e.g., c("L", "R", "X"))
k	Integer, parameter for reverse operations

Value

Numeric matrix (nrow x n) with transformed states

Examples

```
if (cayley_gpu_available()) {
  mat <- matrix(c(1,2,3,4,5, 5,4,3,2,1), nrow = 2, byrow = TRUE)
  result <- apply_operations_batch_gpu(mat, c("1", "3"), k = 4)
}
```

bidirectional_bfs	<i>Bidirectional BFS Shortest Path</i>
-------------------	--

Description

Finds the shortest path between two permutation states using bidirectional breadth-first search. Expands from both the start and goal states simultaneously, meeting in the middle.

Usage

```
bidirectional_bfs(n, state1, state2, max_level, moves, k)
```

Arguments

n	Integer, size of the permutation
state1	Integer vector, start state
state2	Integer vector, goal state
max_level	Integer, maximum BFS depth in each direction
moves	Character vector, allowed operations (e.g., c("1", "2", "3"))
k	Integer, parameter for reverse operations

Value

Character vector of operations forming the shortest path, or NULL if no path found within max_level

Examples

```
# Find path between two small states
path <- bidirectional_bfs(5, 1:5, c(2, 3, 4, 5, 1), max_level = 5,
                        moves = c("1", "2", "3"), k = 3)
path
```

breakpoint_distance	<i>Breakpoint Distance Between Two States</i>
---------------------	---

Description

Counts the number of positions where consecutive elements differ by more than 1 (breakpoints). Particularly effective for TopSpin puzzles where operations shift blocks and flip prefixes.

Usage

```
breakpoint_distance(start_state, target_state)
```

Arguments

start_state Integer vector, first state
target_state Integer vector, second state

Value

Integer, the number of breakpoints

Examples

breakpoint_distance(1:5, 5:1)
breakpoint_distance(1:5, 1:5)

build_permutation_matrix
<i>Build permutation matrix for a single operation</i>

Description

Creates an $n \times n$ permutation matrix (column-major, F32) that represents a single operation. When multiplied: $\text{new_state} = \text{state} \%*\% P$.

Usage

build_permutation_matrix(op, n, k)

Arguments

op Character, operation code ("L"/"1", "R"/"2", "X"/"3")
n Integer, state length
k Integer, reverse prefix length

Value

Numeric vector of length $n*n$ (column-major permutation matrix)

`calculate_angular_distance_z`

Angular Distance Between Two Celestial Points

Description

Computes the angular distance on the celestial sphere between two points given as coordinate lists (each with a z component).

Usage

```
calculate_angular_distance_z(result1, result2)
```

Arguments

<code>result1</code>	List with component z (complex number)
<code>result2</code>	List with component z (complex number)

Value

Numeric, angular distance in radians

Examples

```
c1 <- convert_LRX_to_celestial(10, 5, 3)
c2 <- convert_LRX_to_celestial(1, 1, 2)
calculate_angular_distance_z(c1, c2)
```

`calculate_differences` *Calculate Manhattan Distances for All States*

Description

Computes the Manhattan distance from a reference state to every row in a table of reachable states, adds a difference column, and sorts by it.

Usage

```
calculate_differences(
  start_state,
  reachable_states_start,
  method = "manhattan",
  use_gpu = FALSE
)
```

Arguments

start_state	Integer vector, the reference state
reachable_states_start	Data frame with V-columns
method	Character, distance method (currently only "manhattan")
use_gpu	Logical, use GPU acceleration via ggmlR if available (default FALSE)

Value

Data frame sorted by difference (ascending)

Examples

```
df <- data.frame(V1 = c(1, 2), V2 = c(2, 1))
calculate_differences(c(1, 2), df)
```

calculate_midpoint_z *Midpoint Between Two Celestial Coordinates*

Description

Computes the midpoint on the celestial sphere between two coordinate sets by averaging Cartesian unit-sphere positions and re-projecting.

Usage

```
calculate_midpoint_z(coords1, coords2)
```

Arguments

coords1	List with theta, phi, omega_conformal (and optionally nL, nR, nX)
coords2	List with theta, phi, omega_conformal (and optionally nL, nR, nX)

Value

List with theta, phi, z, z_bar, omega_conformal, nL, nR, nX

Examples

```
c1 <- convert_LRX_to_celestial(10, 5, 3)
c2 <- convert_LRX_to_celestial(1, 1, 2)
mid <- calculate_midpoint_z(c1, c2)
mid$theta
```

cayley_bfs_full

*Full Breadth-First Search Over the Cayley Graph***Description**

Explores every state reachable from `start_state` by repeatedly applying the allowed operations, recording the graph distance of each state from the start. Unlike [sparse_bfs](#), no pruning is applied: the whole reachable component is enumerated.

Usage

```
cayley_bfs_full(start_state, k, moves = c("L", "R", "X"))
```

Arguments

<code>start_state</code>	Integer vector, the state to explore from
<code>k</code>	Integer, parameter for the reverse-prefix operation
<code>moves</code>	Character vector of allowed operations, e.g. <code>c("L", "R", "X")</code> or <code>c("1", "2", "3")</code> (default: all three)

Details

Celestial coordinates are a property of a path rather than of a state, since the same state can be reached by many different operation sequences. The `nL`, `nR`, `nX` counters reported here are those of the shortest path BFS happened to find first, and `theta`, `phi`, `omega` are derived from them via [convert_LRX_to_celestial](#).

Value

Data frame with one row per reachable state:

<code>state_str</code>	State as an underscore-separated key
<code>dist</code>	Graph distance from <code>start_state</code>
<code>nL</code> , <code>nR</code> , <code>nX</code>	Operation counts along the BFS shortest path
<code>theta</code> , <code>phi</code> , <code>omega</code>	Celestial coordinates derived from those counts

See Also

[cayley_graph_diameter](#), [sparse_bfs](#)

Examples

```
d <- cayley_bfs_full(1:5, k = 3)
nrow(d)
table(d$dist)
```

cayley_gpu_available	<i>Check if GPU acceleration is available</i>
----------------------	---

Description

Checks whether ggmlR is installed and Vulkan GPU is present.

Usage

```
cayley_gpu_available()
```

Value

Logical

Examples

```
cayley_gpu_available()
```

cayley_gpu_free	<i>Free GPU backend resources</i>
-----------------	-----------------------------------

Description

Free GPU backend resources

Usage

```
cayley_gpu_free()
```

Value

Invisible NULL

cayley_gpu_init	<i>Initialize GPU backend</i>
-----------------	-------------------------------

Description

Lazily initializes the Vulkan backend. Safe to call multiple times.

Usage

```
cayley_gpu_init(device = 0L, force = FALSE)
```

Arguments

device	Integer, Vulkan device index (0-based)
force	Logical, force re-initialization

Value

Invisible backend pointer

Examples

```
if (cayley_gpu_available()) {  
  cayley_gpu_init()  
}
```

cayley_gpu_status	<i>Get GPU status information</i>
-------------------	-----------------------------------

Description

Get GPU status information

Usage

```
cayley_gpu_status()
```

Value

List with availability, device info, and backend status

Examples

```
cayley_gpu_status()
```

cayley_graph_diameter *Cayley Graph Diameter and Maximally Distant State Pairs*

Description

Computes the diameter of the Cayley graph component reachable from `start_state`, together with the pairs of states realising it and the eccentricity of each vertex.

Usage

```
cayley_graph_diameter(
    start_state,
    k,
    moves = c("L", "R", "X"),
    method = c("all_pairs", "from_start"),
    max_pairs = Inf,
    verbose = FALSE
)
```

Arguments

<code>start_state</code>	Integer vector, the state to explore from
<code>k</code>	Integer, parameter for the reverse-prefix operation
<code>moves</code>	Character vector of allowed operations (default: <code>c("L","R","X")</code>)
<code>method</code>	Either "all_pairs" (default, exact) or "from_start" (single BFS, exact only for vertex-transitive graphs)
<code>max_pairs</code>	Numeric, maximum number of pairs to materialise in <code>pairs_df</code> (default Inf). <code>n_pairs</code> always reports the honest total.
<code>verbose</code>	Logical; if TRUE, prints progress during the sweep

Details

Two methods are available. "all_pairs" runs a BFS from every vertex and yields the true diameter and every diametral pair; its cost grows as the number of vertices times the cost of one BFS, which in practice limits it to permutations of roughly size 8 or below. "from_start" runs a single BFS and reports the eccentricity of `start_state` and the pairs (`start_state`, `v`) realising it; this equals the diameter only when the graph is vertex-transitive, but it scales to much larger graphs.

Value

List containing:

<code>diameter</code>	Integer, the graph diameter (or start eccentricity)
<code>n_vertices</code>	Number of reachable states
<code>n_pairs</code>	Total number of maximally distant pairs found

truncated	Logical, TRUE if pairs_df was capped by max_pairs
pairs_df	Data frame of maximally distant pairs, with the celestial coordinates of both endpoints (from_* and to_* columns)
ecc	Data frame of per-vertex eccentricities (all NA except the start vertex when method = "from_start")
bfs	The full BFS data frame from start_state, as returned by cayley_bfs_full
dist_hist	Data frame of distance-from-start counts
method	The method actually used

See Also

[cayley_bfs_full](#)

Examples

```
res <- cayley_graph_diameter(1:5, k = 3)
res$diameter
head(res$pairs_df)
```

check_duplicates	<i>Find Duplicate States Between Two Tables</i>
------------------	---

Description

Identifies states that appear in both tables by comparing V-columns. Used for finding intersections between forward and backward searches.

Usage

```
check_duplicates(df1, df2)
```

Arguments

df1	Data frame (first set of states)
df2	Data frame (second set of states)

Value

Data frame of duplicate states with a source column, or NULL if none

Examples

```
df1 <- data.frame(V1 = c(1, 2), V2 = c(2, 1))
df2 <- data.frame(V1 = c(2, 3), V2 = c(1, 2))
check_duplicates(df1, df2)
```

`compose_permutation_matrix`*Compose permutation matrices for a sequence of operations*

Description

Multiplies individual permutation matrices into one combined matrix.

Usage

```
compose_permutation_matrix(operations, n, k)
```

Arguments

<code>operations</code>	Character vector of operation codes
<code>n</code>	Integer, state length
<code>k</code>	Integer, reverse prefix length

Value

Numeric vector of length $n*n$ (combined permutation matrix, column-major)

`convert_digits`*Convert String to Integer Vector of Digits*

Description

Parses a string of digits or space-separated numbers into an integer vector. Useful for converting operation sequences or state representations.

Usage

```
convert_digits(s)
```

Arguments

<code>s</code>	Character string. Either a string of single digits (e.g., "123") or space-separated numbers (e.g., "1 2 3" or "10 11 12").
----------------	--

Value

Integer vector of parsed numbers

Examples

```
convert_digits("123")
convert_digits("1 5 4 3 2")
convert_digits("10 11 12 13")
```

 convert_LRX_to_celestial

Convert LRX Counts to Celestial Coordinates

Description

Maps cumulative operation counts (Left, Right, Reverse) to spherical celestial coordinates via stereographic projection.

Usage

```
convert_LRX_to_celestial(nL, nR, nX)
```

Arguments

nL	Integer, cumulative count of left shift operations
nR	Integer, cumulative count of right shift operations
nX	Integer, cumulative count of reverse operations

Value

List with components:

z	Complex number, stereographic projection coordinate
z_bar	Complex conjugate of z
theta	Numeric, zenith angle (from X axis)
phi	Numeric, azimuthal angle (in LR plane)
omega_conformal	Numeric, conformal energy (magnitude of momentum vector)

Examples

```
coords <- convert_LRX_to_celestial(10, 5, 3)
coords$theta
coords$phi
```

Description

Builds the convex hull of a set of points in three dimensions and reports its triangular faces, surface area and volume. Implemented directly so the package keeps its single dependency on Rcpp; for the handful of points a landmark study produces, the incremental algorithm below is more than fast enough.

Usage

```
convex_hull_3d(pts, tol = 1e-09)
```

Arguments

pts	Numeric matrix with three columns, one row per point.
tol	Numeric, distance below which a point counts as lying on a plane. Scaled by the spread of the cloud, so the default suits any magnitude.

Details

The construction is the classical incremental one. A non-degenerate starting tetrahedron is found first, then the remaining points are added one at a time: every face the new point can "see" (the point lies on the outer side of its plane) is removed, and the boundary of the hole left behind – the horizon – is joined to the new point. Points that see no face are already inside and are skipped.

Volume is the sum of the signed tetrahedra spanned by each face and an interior reference point; because every face is oriented outwards, the signs agree and the total is the enclosed volume.

Value

A list with components:

faces	Integer matrix, one row per triangular face, holding indices into pts in counter-clockwise order seen from outside
vertices	Integer vector, the rows of pts that are hull vertices
area	Numeric, total surface area
volume	Numeric, enclosed volume
degenerate	Logical, TRUE when the points are coplanar or fewer than four were supplied, in which case volume is 0

Examples

```
cube <- as.matrix(expand.grid(c(0, 1), c(0, 1), c(0, 1)))
h <- convex_hull_3d(cube)
h$area      # 6
h$volume    # 1
```

create_state_store	<i>Create a New State Store</i>
--------------------	---------------------------------

Description

Creates a C++ StateStore object for compact, incremental storage of permutation states with hash-indexed lookup.

Usage

```
create_state_store(perm_length, init_capacity = 10000L)
```

Arguments

perm_length	Integer, length of each permutation state
init_capacity	Integer, initial capacity (default 10000)

Value

External pointer to StateStore (XPtr)

Examples

```
store <- create_state_store(6L)
state_store_size(store)
```

cycle_shortcut	<i>Shorten a Path by Cutting Across Cycles</i>
----------------	--

Description

Shortens an existing path by looking for cycles that leave it and rejoin it further along. A cycle here is a combo word (say "1231") applied round and round until the state returns to where the cycle started – the same construction [find_path_iterative](#) searches with. If some state along that loop also occurs later in the path, the stretch between the two meeting points can be replaced by the stretch of the loop, and whenever the loop covers it in fewer operations the path gets shorter.

Usage

```
cycle_shortcut(
  path,
  start_state,
  k,
  n_points = 20L,
  moves = c("1", "2", "3"),
  combo_length = 20L,
```

```

    n_samples = 200L,
    n_top = 5L,
    sort_by = c("longest", "most_unique"),
    max_cycle_len = 20000L,
    n_threads = NULL,
    verbose = FALSE
)

```

Arguments

path	Character vector of operations ("1"/"2"/"3")
start_state	Integer vector, the state the path starts from
k	Integer, length of the reverse-prefix (flipper) operation
n_points	Integer, how many places along the path to try. Points are spread evenly with a random jitter; cost grows linearly with this, path coverage does not (default 20)
moves	Character vector, operations the combos are drawn from (default c("1", "2", "3"))
combo_length	Integer, length of each sampled combo word (default 20)
n_samples	Integer, combos sampled per point before ranking (default 200)
n_top	Integer, top-ranked combos actually spun into cycles per point (default 5)
sort_by	Character vector, ranking criteria for the combos, applied in order. One or more of "longest", "shortest", "most_unique", "least_unique", "most_repeated", "least_repeated", as in find_best_random_combinations . Pass NULL to skip ranking altogether and simply take the first n_top combos as drawn: ranking has to unroll every one of the n_samples candidates to score it, so turning it off is markedly cheaper, at the price of picking blind. Default c("longest", "most_unique")
max_cycle_len	Integer, cap on how far a single cycle is unrolled. Some combos close only after hundreds of thousands of steps. Capped at the current path length regardless: a detour longer than the whole path can never win (default 20000)
n_threads	Integer, OpenMP threads used to score and search combos. Defaults to two below the core count, leaving room for the rest of the machine; pass a number to fix it, or NA to use whatever OpenMP would pick by itself. Points remain sequential whatever this is set to – each searches the path the previous one shortened.
verbose	Logical, print progress (default FALSE)

Details

This complements [short_path_bfs](#) rather than replacing it. That one sweeps the BFS neighbourhood a few steps deep, so it finds cuts between nearby points; a cycle wanders hundreds of steps from where it started and reaches rejoin points BFS depth cannot see. Running one after the other is reasonable.

Points are processed one at a time, and a cut is applied the moment it is found: every later point is then searched against the already-shortened path. This is why the points are carried as states rather than as positions – a cut renumbers everything downstream of it, but a state is either still somewhere in the path or it is not. Points swallowed by an earlier cut are simply skipped.

Value

List with components:

path	Character vector, the shortened path (the original if nothing was found or verification failed)
original_length	Integer, length of the input path
new_length	Integer, length of the returned path
savings	Integer, operations removed
n_cuts	Integer, how many cuts were applied
cuts	Data frame of the applied cuts, in the order they were applied: from, to, cycle_len, gain (positions refer to the path as it stood when that cut was made)

See Also

[short_path_bfs](#), [human_algorithm_to](#), [find_path_iterative](#)

Examples

```
set.seed(1)
s <- generate_state(20, k = 4, n_moves = 300)
res <- human_algorithm_to(s, k = 4, simplify = FALSE)
cut <- cycle_shortcut(res$path, s, k = 4, n_points = 5)
cut$savings
```

distance_methods

Distance Methods for Bridge Selection

Description

The registry of methods [find_path_iterative](#) may use to score candidate states when picking bridges. Each method is a function `f(states, target, k)` returning one score per row of states; lower is better. The search takes the lowest-scoring candidate, breaking ties on the smaller step number.

Usage

```
cayley_distance_methods()
```

```
cayley_distance(method)
```

Arguments

method	Character, name of a registered method
--------	--

Details

manhattan Sum of absolute differences to the target. The default, and the only behaviour available before methods became pluggable.

breakpoints Number of adjacency violations relative to the target – positions where consecutive values are not consecutive in the target’s frame. Often a better guide than raw displacement.

human How far the state is from being solved *the way a person solves it*: the number of tiles still missing from the sorted run. Scores against the identity 1:n and ignores target, so it applies only to searches heading for the sorted ring. Ties break on the gap phase 1 works to close. See [find_best_match_human](#).

Every method is a function of (states, target, k): a matrix with one candidate state per row, the state being approached, and the flipper width (used by human, ignored by the rest). It returns one score per row, lower being better, with NA marking a candidate it rejects. Those are the arguments of the returned method, not of the two functions here.

Value

cayley_distance_methods() returns the registered names; cayley_distance() returns the method itself, as a function.

See Also

[find_path_iterative](#), [human_phase1_rank](#)

Examples

```
m <- rbind(c(1L, 2L, 3L, 4L), c(4L, 3L, 2L, 1L))
cayley_distance_methods()           # registered method names
cayley_distance("manhattan")(m, 1:4, 4)
```

enclosing_hull_3d

Enclosing Hull Through Every Point

Description

Builds a closed triangulated surface whose vertices are *all* the given points, not just the ones on the convex boundary.

Usage

```
enclosing_hull_3d(pts, tol = 1e-09)
```

Arguments

pts Numeric matrix with three columns, one row per point.

tol Numeric, tolerance passed through to [convex_hull_3d](#).

Details

`convex_hull_3d` returns the smallest convex body containing the cloud, so any point strictly inside it is not a vertex of the result. When the point set is meant to be read as the corners of one figure, that is the wrong answer: some of the corners go missing. This function starts from the convex hull and then, for each interior point in turn, replaces the triangle whose plane it sits closest under with three triangles meeting at that point. The surface is dented inwards there, so the body stops being convex, but every supplied point ends up on it.

Because the surface is no longer convex, the volume is computed as the signed sum of tetrahedra over the oriented faces, which stays correct for any closed surface that does not intersect itself.

Value

A list with the same components as `convex_hull_3d` plus

pushed	Integer vector, the points that had to be pulled onto the surface, i.e. those that were interior to the convex hull
--------	---

See Also

`convex_hull_3d`

Examples

```
set.seed(1)
p <- rbind(as.matrix(expand.grid(c(0, 1), c(0, 1), c(0, 1))), c(0.5, 0.5, 0.5))
h <- enclosing_hull_3d(p)
length(h$vertices) # 9: the interior point is a vertex too
```

find_best_match_human *Score Candidate States the Way a Person Solves*

Description

Distance is how much of the sorted run is still missing: `n - run_length`. Within a run length, ties break on the gap phase 1 works to close, so among states with an equal run the one closest to placing its next value scores lower.

Usage

```
find_best_match_human(states, target, k)
```

Arguments

states	Integer matrix, one candidate state per row
target	Integer vector, ignored; accepted for interface uniformity
k	Integer, flipper width

Details

Scoring is against the identity 1:n and target is ignored. That is deliberate, not an oversight: run_length only means something when the goal is the sorted ring. Relabelling into an arbitrary target's frame was tried and performed worse – the side of a two-ended search that grows from an unstructured state ends up judged against a goal phase 1 cannot read. Use this method for searches heading for 1:n, as the tail search after phase 1 does.

Registered as distance method "human"; see [distance_methods](#).

Scoring runs in C++ (human_distance_cpp); the whole candidate set is scored in one call.

Value

Numeric vector of scores, one per row; lower is better

See Also

[distance_methods](#), [human_phase1_rank](#)

find_best_random_combinations

Find Best Random Operation Sequences

Description

Generates random sequences of operations and evaluates their cycle lengths to find sequences that produce the best cycles in the Cayley graph. Uses C++ with OpenMP for parallel evaluation of combinations.

Usage

```
find_best_random_combinations(
  moves,
  combo_length,
  n_samples,
  n_top,
  start_state,
  k,
  sort_by = c("longest", "most_unique")
)
```

Arguments

moves	Character vector of allowed operation symbols (e.g., c("1", "2", "3") or c("L", "R", "X"))
combo_length	Integer, length of each operation sequence to test
n_samples	Integer, number of random sequences to generate and test

<code>n_top</code>	Integer, number of top results to return
<code>start_state</code>	Integer vector, initial permutation state
<code>k</code>	Integer, parameter for reverse operations
<code>sort_by</code>	Character vector of sorting criteria, applied in order. Available criteria: "longest" Prefer longer cycles (descending <code>total_moves</code>) "shortest" Prefer shorter cycles (ascending <code>total_moves</code>) "most_unique" Prefer more unique states (descending <code>unique_states_count</code>) "least_unique" Prefer fewer unique states (ascending <code>unique_states_count</code>) "most_repeated" Prefer higher repetition ratio <code>total_moves/unique_states_count</code> (descending) "least_repeated" Prefer lower repetition ratio (ascending) Default: <code>c("longest", "most_unique")</code> (original behavior).

Value

Data frame with columns:

<code>combo_number</code>	Integer sequence number
<code>combination</code>	String representation of the operation sequence
<code>total_moves</code>	Cycle length for this sequence
<code>unique_states_count</code>	Number of unique states visited in the cycle
<code>repetition_ratio</code>	Ratio <code>total_moves / unique_states_count</code>

Examples

```
# Default: longest cycles
best <- find_best_random_combinations(
  moves = c("1", "2", "3"),
  combo_length = 10,
  n_samples = 50,
  n_top = 5,
  start_state = 1:10,
  k = 4
)

# Short cycles with many unique states
best2 <- find_best_random_combinations(
  moves = c("1", "2", "3"),
  combo_length = 10,
  n_samples = 50,
  n_top = 5,
  start_state = 1:10,
  k = 4,
  sort_by = c("shortest", "most_unique")
)
```

find_closest_to_coords

Find Closest State to Target Coordinates

Description

Searches a table of reachable states for the state whose celestial coordinates are closest to a target coordinate set.

Usage

```
find_closest_to_coords(reachable_states, target_coords, v_cols)
```

Arguments

reachable_states	Data frame with columns theta, phi, omega_conformal, and V-columns for state
target_coords	List with component z (complex number)
v_cols	Character vector of V-column names

Value

Single-row data frame of the closest state (with angular_distance column added)

Examples

```
# Typically used with output from get_reachable_states
# find_closest_to_coords(states_df, target_coords, paste0("V", 1:n))
```

find_combination_in_states

Find a State in Reachable States Table

Description

Searches for a specific permutation state in a reachable states table and returns the first matching row with metadata.

Usage

```
find_combination_in_states(reachable_states_start, search_state)
```

Arguments

reachable_states_start	Data frame with V-columns and metadata
search_state	Integer vector, the state to search for

Value

Data frame row with state and metadata columns, or NULL if not found

Examples

```
df <- data.frame(V1 = c(1, 2), V2 = c(2, 1), operation = c("1", "2"),
                 step = c(1, 2), combo_number = c(1, 1))
find_combination_in_states(df, c(2, 1))
```

find_path_bfs	<i>Find Path via BFS Highways</i>
---------------	-----------------------------------

Description

Builds BFS highway trees from start and final states, finds the closest pair of hub states (one from each highway), then uses find_path_iterative to connect them. Assembles the full path: bfs(start -> hub_s) + iterative(hub_s -> hub_f) + inverted_bfs(final -> hub_f)

Usage

```
find_path_bfs(
  start_state,
  final_state,
  k,
  bfs_levels = 500L,
  bfs_n_hubs = 7L,
  bfs_n_random = 3L,
  highway_distance_method = "manhattan",
  iterative_distance_method = highway_distance_method,
  verbose = TRUE,
  ...
)
```

Arguments

start_state	Integer vector, the starting permutation state
final_state	Integer vector, the target permutation state
k	Integer, parameter for reverse operations
bfs_levels	Integer, depth of sparse BFS from each side (default 500)
bfs_n_hubs	Integer, top-degree nodes per BFS level (default 7)
bfs_n_random	Integer, random nodes per BFS level (default 3)
highway_distance_method	Character, "manhattan", "breakpoints" or "human" (default "manhattan"). Used to pair the two BFS highway ends. "human" scores a state by how much of the sorted run 1..r it has built, so it is only meaningful when final_state is 1:n.

iterative_distance_method	Character, the method find_path_iterative uses to pick bridges when searching between the hubs. Defaults to highway_distance_method. Kept separate because the two steps compare different things: highway pairing measures the gap between the two ends, while the search targets a hub, not 1:n, so "human" fits it only when that hub is 1:n.
verbose	Logical, print progress (default TRUE)
...	Additional arguments passed to find_path_iterative

Value

List with path, found, cycles, bfs_info

find_path_iterative	<i>Iterative Path Finder Between Permutation States</i>
---------------------	---

Description

Finds a path between two permutation states using iterative cycle expansion. Generates random operation sequences, analyzes their cycles, and looks for intersections between forward (from start) and backward (from final) state sets. Uses bridge states to progressively narrow the search space.

Usage

```
find_path_iterative(
  start_state,
  final_state,
  k,
  moves = c("1", "2", "3"),
  combo_length = 20,
  n_samples = 200,
  n_top = 10,
  max_iterations = 10,
  potc = 1,
  ptr = 10,
  opd = FALSE,
  reuse_combos = FALSE,
  keep_states = FALSE,
  one_sided = FALSE,
  distance_method = "manhattan",
  sort_by = c("longest", "most_unique"),
  verbose = TRUE
)
```

Arguments

start_state	Integer vector, the starting permutation state
final_state	Integer vector, the target permutation state
k	Integer, parameter for reverse operations
moves	Character vector, allowed operations (default c("1", "2", "3"))
combo_length	Integer, length of random operation sequences (default 20)
n_samples	Integer, number of random sequences to test per iteration (default 200)
n_top	Integer, number of top sequences to analyze fully (default 10)
max_iterations	Integer, maximum number of search iterations (default 10)
potc	Numeric in (0,1], fraction of cycle states to keep (default 1)
ptr	Integer, max intersections to process per iteration (default 10)
opd	Logical, if TRUE filters states to only combos containing bridge state (default FALSE)
reuse_combos	Logical, if TRUE generates random combos only once per side (cycle 1) and reuses them in subsequent cycles. Saves time but reduces diversity (default FALSE)
keep_states	Logical, if TRUE every cycle's states stay in the store (memory grows with the number of cycles). If FALSE (default) each cycle's states are dropped once its bridge is chosen, so memory stays flat: only the current cycle plus the operation segment recorded on each bridge is kept. Intersections are then found only between states of the same cycle, never across cycles.
one_sided	Logical, if TRUE the final side is expanded only in cycle 1 and then left frozen, with the search advancing from the start side alone (default FALSE)
distance_method	Character, method for comparing states during bridge selection. One of "manhattan" (sum of absolute differences) or "breakpoints" (number of adjacency violations). Default "manhattan".
sort_by	Character vector of sorting criteria for combo selection. See find_best_random_combinations for details. Default: c("longest", "most_unique").
verbose	Logical, if TRUE prints progress messages (default TRUE)

Details

Uses a compact C++ StateStore backend for O(1) incremental hash-indexed state storage, eliminating quadratic memory growth from repeated rbind.

Value

List containing:

path	Character vector of operations, or NULL if not found
found	Logical, whether a path was found
cycles	Number of iterations used

```

selected_info    Details about the selected intersection
bridge_states_start
                  List of forward bridge states
bridge_states_final
                  List of backward bridge states

```

Examples

```

# Small example
set.seed(42)
start <- 1:6
final <- c(3L, 1L, 2L, 6L, 4L, 5L)
# result <- find_path_iterative(start, final, k = 3, max_iterations = 5)

```

generate_state	<i>Generate Reachable Random State</i>
----------------	--

Description

Generates a random state reachable from 1:n by applying random operations (L, R, X). Guarantees the result is in the same connected component as the starting state.

Usage

```

generate_state(
  n,
  k = n,
  n_moves = 25L,
  moves = c("1", "2", "3"),
  max_attempts = 100L
)

```

Arguments

n	Integer, the size of the permutation
k	Integer, parameter for reverse_prefix operation
n_moves	Integer, number of random operations to apply (default 25)
moves	Character vector, allowed operations (default c("1", "2", "3"))
max_attempts	Integer, maximum attempts to generate a non-identity state (default 100)

Value

Integer vector representing a reachable permutation state

Examples

```

set.seed(42)
generate_state(10, k = 4)
generate_state(10, k = 4, n_moves = 100)

```

```
generate_unique_states_df
```

Generate Data Frame of Unique Random States

Description

Generates a data frame with unique random permutation states.

Usage

```
generate_unique_states_df(n, n_rows)
```

Arguments

n	Integer, size of each permutation state
n_rows	Integer, number of unique states to generate

Value

Data frame with n_rows rows and columns V1, V2, ..., Vn

Examples

```
set.seed(42)
df <- generate_unique_states_df(5, 10)
head(df)
```

```
get_reachable_states
```

Find Cycle in Permutation Group

Description

Explores the Cayley graph starting from an initial state and applying a sequence of operations repeatedly until returning to the start state. Returns detailed information about all visited states, the cycle structure, and celestial LRX coordinates.

Usage

```
get_reachable_states(start_state, allowed_positions, k, verbose = FALSE)
```

Arguments

start_state	Integer vector, the initial permutation state
allowed_positions	Character vector, sequence of operations to repeat
k	Integer, parameter for reverse operations
verbose	Logical; if TRUE, prints progress and cycle information (default FALSE)

Value

List containing:

states	List of all visited states
reachable_states_df	Data frame with states, operations, steps, and celestial coordinates
operations	Vector of operations applied
coords	List of celestial coordinate objects per step
nL_total	Total left shifts
nR_total	Total right shifts
nX_total	Total reverse operations
total_moves	Total number of moves in the cycle
unique_states_count	Number of unique states visited
cycle_info	Summary string with cycle statistics

Examples

```
result <- get_reachable_states(1:10, c("1", "3"), k = 4)
writeLines(result$cycle_info)
```

```
get_reachable_states_light
```

Find Cycle Length (Lightweight Version)

Description

Fast version of cycle detection that only returns cycle length and unique state count without storing all intermediate states. Useful for testing many operation sequences efficiently. Implemented in C++ for performance.

Usage

```
get_reachable_states_light(start_state, allowed_positions, k)
```

Arguments

start_state	Integer vector, the initial permutation state
allowed_positions	Character vector, sequence of operations to repeat
k	Integer, parameter for reverse operations

Value

List containing:

total_moves	Total number of moves to return to start state
unique_states_count	Number of unique states in the cycle

Examples

```
result <- get_reachable_states_light(1:10, c("1", "3"), k = 4)
cat("Cycle length:", result$total_moves, "\n")
cat("Unique states:", result$unique_states_count, "\n")
```

human_algorithm

Solve a State with the Human TopSpin Algorithm

Description

Reproduces the way a person solves TopSpin by hand. A sorted run is grown one value at a time: for each new value m the ring is manoeuvred until m sits exactly k positions after $m-1$, so that a single reverse-prefix drops m directly behind $m-1$. Auxiliary flips are restricted to windows lying wholly inside the unsorted arc, so the run is never disturbed.

Usage

```
human_algorithm(start_state, final_state = NULL, k = 4L, simplify = TRUE)
```

Arguments

start_state	Integer vector, the starting permutation state
final_state	Integer vector, the target state. Defaults to $1:n$, i.e. plain sorting.
k	Integer, length of the reverse-prefix (flipper) operation
simplify	Logical, run short_position() on the result

Details

Once the tail is down to eight tiles the insertion move no longer fits, and the tail is finished with two local 3-cycles built from the same operations (XLXLXR and LXRXLXL, each with a compensating rotation). Their conjugates generate the full alternating group on the tail, so any even arrangement is reachable by table lookup rather than search.

Because 3-cycles are even permutations, odd tail arrangements are out of their reach. In that case a single flip is fired across the block boundary and the run is rebuilt: moving one tile between block and tail changes the parity of the split, after which the table applies.

Value

List with components:

found	Logical, whether the target was reached
path	Character vector of operations ("1"/"2"/"3")
length	Integer, number of operations

Examples

```
set.seed(1)
s <- generate_state(20, k = 4, n_moves = 50)
res <- human_algorithm(s, k = 4)
res$found
```

human_algorithm_to	<i>Human Algorithm Path Between Two Arbitrary States</i>
--------------------	--

Description

Finds a path from `start_state` to `target_state` with the same human TopSpin method as [human_algorithm](#), but without routing through the identity state.

Usage

```
human_algorithm_to(start_state, target_state = NULL, k = 4L, simplify = TRUE)
```

Arguments

<code>start_state</code>	Integer vector, the starting permutation state
<code>target_state</code>	Integer vector, the target permutation state. Defaults to <code>1:n</code> , in which case this is exactly human_algorithm .
<code>k</code>	Integer, length of the reverse-prefix (flipper) operation
<code>simplify</code>	Logical, run <code>short_position()</code> on the result

Details

`human_algorithm` reaches an arbitrary target by solving both endpoints to `1:n` and concatenating the first path with the inverse of the second, so the word is roughly twice as long as it needs to be. Here the problem is instead relabelled: because the three operations permute *positions* and treat the values as inert labels, renaming every value `v` to its position in `target_state` turns "reach `target_state`" into "reach `1:n`". The solver then runs once on the relabelled state and the resulting word applies unchanged to the original one.

Formally, with `inv` the inverse permutation of the target (`inv[target_state] == seq_len(n)`), the relabelled state is `inv[start_state]`. Any word `P` sorting it satisfies `P(inv[start_state]) == 1:n`, and since `P` acts on positions it commutes with the elementwise relabelling, giving `P(start_state) == target_state`.

Value

List with components:

found	Logical, whether the target was reached
path	Character vector of operations ("1"/"2"/"3")
length	Integer, number of operations

See Also

[human_algorithm](#)

Examples

```
set.seed(1)
s <- generate_state(20, k = 4, n_moves = 50)
t <- generate_state(20, k = 4, n_moves = 50)
res <- human_algorithm_to(s, t, k = 4)
res$found
```

human_phase1_navigate *Follow the Phase 1 Navigator to the Tail*

Description

Repeatedly applies the move [human_phase1_rank](#) prefers, growing the sorted run until phase 1 runs out of applicable moves – that is, until only the tail is left. The tail itself is not touched; finishing it needs either the 3-cycles of [human_algorithm](#) or a search such as [find_path_iterative](#).

Usage

```
human_phase1_navigate(state, k = 4L, max_steps = 2000L, trace = FALSE)
```

Arguments

state	Integer vector, the state to navigate from
k	Integer, length of the reverse-prefix (flipper) operation
max_steps	Integer, safety cap on navigator steps (default 2000)
trace	Logical, collect a per-step data.frame (default FALSE)

Details

The walk is greedy: at each step the single best-ranked candidate is taken.

Value

List with components:

state	Integer vector, the state reached
path	Character vector of operations applied
run	Integer, sorted run length reached
trace	data.frame of per-step progress, or NULL

See Also

[human_phase1_rank](#), [human_algorithm](#)

Examples

```
set.seed(42)
s <- generate_state(20, k = 4, n_moves = 200)
nav <- human_phase1_navigate(s, k = 4)
nav$run
```

human_phase1_rank	<i>Rank Candidate Moves by the Phase 1 Criterion</i>
-------------------	--

Description

Exposes phase 1 of [human_algorithm](#) as a navigator: instead of committing to a move, it reports the moves phase 1 would consider from the given state, each scored. A search can use this to choose its own direction while still being guided by the human method.

Usage

```
human_phase1_rank(state, k = 4L, sorted = TRUE)
```

Arguments

state	Integer vector, a permutation state
k	Integer, length of the reverse-prefix (flipper) operation
sorted	Logical, return rows in phase 1 preference order (default TRUE)

Details

The candidates are *composite* moves – "rotate the ring so the flipper covers a chosen window, then flip" – which is the unit phase 1 actually works in. Ranking the three raw operations instead would give no signal: a single rotation changes neither the run nor the gap. One candidate is produced per window offset, plus the placing move itself when the gap already equals k. Windows that would overlap the finished run are dropped, exactly as in phase 1.

With sorted = TRUE the rows come back in phase 1's own order of preference: run descending, then gap_cost ascending, then the shorter word.

An empty data frame is returned once the run has grown far enough that only the tail is left: the insertion move no longer fits there, and the tail needs the 3-cycles of phase 2 instead.

Value

A data.frame with one row per candidate move:

ops	Character, the operation word, comma-separated ("1"/"2"/"3")
len	Integer, number of operations in the word
run	Integer, run_length after the move; higher is better
gap_cost	Integer, $ \text{gap}(m-1, m) - k $ afterwards; lower is better
places	Logical, whether the move actually appends a value to the run

See Also

[run_length](#), [human_algorithm](#)

Examples

```
set.seed(1)
s <- generate_state(20, k = 4, n_moves = 50)
human_phase1_rank(s, k = 4)
```

invert_path	<i>Invert a Path of Operations</i>
-------------	------------------------------------

Description

Reverses and inverts a sequence of operations. "1" (shift left) becomes "2" (shift right) and vice versa. "3" (reverse) stays the same.

Usage

```
invert_path(path)
```

Arguments

path	Character vector of operations
------	--------------------------------

Value

Character vector of inverted operations in reverse order

Examples

```
invert_path(c("1", "3", "2"))
invert_path(c("1", "1", "3"))
```

landmark_states

*Generate Landmark States for a Permutation of Size n***Description**

Builds 25 structurally distinct permutations of $1:n$, each defined by a rule rather than by a random draw, so the same construction can be compared across different n . They serve as landmarks (fixed probe points) in the Cayley graph: the distance from the identity to each landmark, measured for several small n where the diameter is known, gives a ratio $d / \text{diameter}$ that can be extrapolated to larger graphs.

Usage

```
landmark_states(n)
```

Arguments

n Integer, permutation size (must be at least 4).

Details

The constructions, in the order returned:

1. **full_reverse** — $\sigma(j) = n + 1 - j$, the maximum number of inversions.
2. **block_swap** — the first half and the last half exchange places. For odd n the middle element stays put.
3. **riffle** — perfect interleaving of the two halves: $1, h + 1, 2, h + 2, \dots$ with $h = \lceil n/2 \rceil$.
4. **envelope** — taken alternately from the two ends towards the centre: $n, 1, n - 1, 2, \dots$
5. **adjacent_swaps** — the full pairing of neighbours: $\text{swap}(1,2), \text{swap}(3,4), \dots$ A trailing odd element is left alone.
6. **broken_cycle** — the n -cycle $2, 3, \dots, n, 1$ with the last two entries exchanged.
7. **zigzag** — all odd values ascending, then all even values descending.
8. **block_rotate3** — the sequence cut into three blocks ABC of deliberately unequal length and reassembled as CAB. Equal blocks would make this a plain rotation, i.e. one L move from the identity.
9. **two_cycles** — two independent cyclic shifts, one inside each half.
10. **shift_reverse** — a left shift by 2 followed by reversing the first 4 elements.
11. **pair_shift** — odd positions take the value two ahead, wrapping round the odd positions only.
12. **reverse_first** — the first half reversed, the second fixed.
13. **reverse_second** — the first half fixed, the second reversed.
14. **spiral** — $1, n, 2, n - 1, \dots$, alternating from the two ends but starting at the bottom.
15. **local_block** — a single block of four rotated in place, every other tile untouched.

16. **single_swap** — one transposition in the middle; the permutation closest to the identity in this set.
17. **faro_in** — the mirror of riffle: the upper half leads the interleaving.
18. **block_reverse_pairs** — the pairs (1,2)(3,4)... kept intact but listed in reverse block order.
19. **doubling** — $\sigma(j) = 2j \bmod (n + 1)$. That is a bijection only for even n ; for odd n the map runs on $1 \dots n-1$ modulo n and the last tile stays put.
20. **shift_third** — a shift by $n/3$ with the displaced block reversed.
21. **double_riffle** — the halves interleaved two elements at a time rather than one.
22. **cycles3** — 3-cycles (1 2 3)(4 5 6)... with any tail fixed.
23. **alt_pairs** — every other pair reversed: swap(1,2), leave (3,4), swap(5,6), ...
24. **cascade** — swapped pairs offset by one as they march along, like falling dominoes.
25. **derangement** — the halves exchanged and then every fixed point displaced, so no tile keeps its own place. Reversing one half instead would only reproduce `reverse_first` rotated by $n/2$.

At $n = 6$ `cycles3` degenerates into `two_cycles`; for every n from 7 upwards all 25 states are distinct.

Value

A `data.frame` with one row per landmark and columns `id`, `name`, `description`, `state_str` (underscore-joined) and `state` (a list column holding the integer vector).

Examples

```
landmark_states(10)$state_str
```

manhattan_distance	<i>Manhattan Distance Between Two States</i>
--------------------	--

Description

Computes the sum of absolute differences between corresponding elements of two permutation states.

Usage

```
manhattan_distance(start_state, target_state)
```

Arguments

<code>start_state</code>	Integer vector, first state
<code>target_state</code>	Integer vector, second state

Value

Numeric, the Manhattan distance

Examples

```
manhattan_distance(1:5, 5:1)
manhattan_distance(1:5, 1:5)
```

manhattan_distance_matrix_gpu

Compute Pairwise Manhattan Distance Matrix on GPU

Description

Computes all pairwise Manhattan distances between two sets of states. Returns an $N1 \times N2$ matrix where entry (i,j) is the Manhattan distance between row i of `states1` and row j of `states2`.

Usage

```
manhattan_distance_matrix_gpu(states1, states2, batch_size = 256L)
```

Arguments

<code>states1</code>	Numeric matrix ($N1 \times n$), first set of states
<code>states2</code>	Numeric matrix ($N2 \times n$), second set of states
<code>batch_size</code>	Integer, number of <code>states2</code> rows to process at once (default 256)

Details

For large matrices, computation is batched over columns of the result to avoid GPU memory overflow.

Value

Numeric matrix ($N1 \times N2$) of Manhattan distances

Examples

```
if (cayley_gpu_available()) {
  s1 <- matrix(c(1,2,3,4,5, 5,4,3,2,1), nrow = 2, byrow = TRUE)
  s2 <- matrix(c(3,3,3,3,3, 1,1,1,1,1), nrow = 2, byrow = TRUE)
  manhattan_distance_matrix_gpu(s1, s2)
}
```

openmp_threads	<i>Number of OpenMP Threads Available</i>
----------------	---

Description

Reports how many threads OpenMP would use by default, which is the core count unless OMP_NUM_THREADS says otherwise. Returns 1 when the package was built without OpenMP.

Usage

```
openmp_threads()
```

Details

Useful for sizing the n_threads argument of [cycle_shortcut](#), whose own default is two below this number.

Value

Integer, the thread count

Examples

```
openmp_threads()
```

reconstruct_bfs_path	<i>Reconstruct path from sparse BFS result</i>
----------------------	--

Description

Traces back from target_key to the root (start state) using the parent_key/child_key edges in the BFS result.

Usage

```
reconstruct_bfs_path(bfs_result, target_key)
```

Arguments

- bfs_result data.frame returned by sparse_bfs()
- target_key Character string — state key to trace back from

Value

Character vector of operations from start to target

reverse_prefix	<i>Reverse First k Elements (with Coordinates)</i>
----------------	--

Description

Reverses the first k elements of the state vector (turnstile operation). Tracks celestial coordinates (LRX momentum).

Usage

```
reverse_prefix(state, k, coords = NULL)
```

Arguments

state	Integer vector representing the current permutation state
k	Integer, number of elements to reverse from the beginning
coords	Optional list of current celestial coordinates. If NULL, starts from zero coordinates.

Value

List with components:

state	Integer vector with first k elements reversed
coords	List of updated celestial coordinates (nL, nR, nX, theta, phi, omega_conformal)

Examples

```
result <- reverse_prefix(1:10, k = 4)
result$state
```

reverse_prefix_simple	<i>Reverse First k Elements (Simple)</i>
-----------------------	--

Description

Simple prefix reversal without coordinate tracking.

Usage

```
reverse_prefix_simple(state, k)
```

Arguments

state	Integer vector representing the current permutation state
k	Integer, number of elements to reverse from the beginning

Value

Integer vector with first k elements reversed

Examples

```
reverse_prefix_simple(1:10, k = 4)
```

run_length	<i>Length of the Sorted Run on the Ring</i>
------------	---

Description

Returns the length of the run 1, 2, . . . , r currently sitting consecutively on the ring, starting wherever value 1 happens to be. This is the quantity phase 1 of [human_algorithm](#) maximises: each insertion move appends one value to the run.

Usage

```
run_length(state)
```

Arguments

state Integer vector, a permutation state

Details

Because the ring is cyclic, the position of value 1 is irrelevant – only the consecutive run following it is counted. A fully sorted state gives n; a state where 2 does not follow 1 gives 1.

Value

Integer, length of the sorted run (0 if value 1 is absent)

See Also

[human_phase1_rank](#), [human_algorithm](#)

Examples

```
run_length(1:20)                    # 20, fully sorted
run_length(c(1:5, 20:16, 6:15))    # 5
```

save_bridge_states	<i>Save Bridge States to CSV</i>
--------------------	----------------------------------

Description

Writes a list of bridge states (each with state and cycle fields) to a CSV file.

Usage

```
save_bridge_states(bridge_states, filename)
```

Arguments

bridge_states	List of lists, each containing state (integer vector) and cycle (integer)
filename	Character, output CSV file path

Value

Invisible NULL. Side effect: writes a CSV file.

Examples

```
bs <- list(  
  list(state = 1:5, cycle = 0),  
  list(state = c(2, 1, 3, 4, 5), cycle = 1)  
)  
# save_bridge_states(bs, tempfile(fileext = ".csv"))
```

select_unique	<i>Select Unique States by V-columns</i>
---------------	--

Description

Removes duplicate rows based on state columns (V1, V2, ..., Vn).

Usage

```
select_unique(df)
```

Arguments

df	Data frame
----	------------

Value

Data frame with unique states

Examples

```
df <- data.frame(V1 = c(1, 1, 2), V2 = c(2, 2, 1), op = c("a", "b", "c"))
select_unique(df)
```

shift_left	<i>Shift State Left (with Coordinates)</i>
------------	--

Description

Performs a cyclic left shift on the state vector, moving the first element to the end. Tracks celestial coordinates (LRX momentum).

Usage

```
shift_left(state, coords = NULL)
```

Arguments

state	Integer vector representing the current permutation state
coords	Optional list of current celestial coordinates. If NULL, starts from zero coordinates.

Value

List with components:

state	Integer vector with elements shifted left by one position
coords	List of updated celestial coordinates (nL, nR, nX, theta, phi, omega_conformal)

Examples

```
result <- shift_left(1:5)
result$state
result$coords

# Chain operations using coords
r1 <- shift_left(1:5)
r2 <- shift_left(r1$state, r1$coords)
r2$coords$nL
```

shift_left_simple	<i>Shift State Left (Simple)</i>
-------------------	----------------------------------

Description

Simple cyclic left shift without coordinate tracking.

Usage

```
shift_left_simple(state)
```

Arguments

state Integer vector representing the current permutation state

Value

Integer vector with elements shifted left by one position

Examples

```
shift_left_simple(1:5)
```

shift_right	<i>Shift State Right (with Coordinates)</i>
-------------	---

Description

Performs a cyclic right shift on the state vector, moving the last element to the front. Tracks celestial coordinates (LRX momentum).

Usage

```
shift_right(state, coords = NULL)
```

Arguments

state Integer vector representing the current permutation state
coords Optional list of current celestial coordinates. If NULL, starts from zero coordinates.

Value

List with components:

state Integer vector with elements shifted right by one position
coords List of updated celestial coordinates (nL, nR, nX, theta, phi, omega_conformal)

Examples

```
result <- shift_right(1:5)
result$state
```

shift_right_simple	<i>Shift State Right (Simple)</i>
--------------------	-----------------------------------

Description

Simple cyclic right shift without coordinate tracking.

Usage

```
shift_right_simple(state)
```

Arguments

state	Integer vector representing the current permutation state
-------	---

Value

Integer vector with elements shifted right by one position

Examples

```
shift_right_simple(1:5)
```

short_path_bfs	<i>Shorten Path via Depth-Limited BFS Hopping</i>
----------------	---

Description

For each position along the path, explores all reachable states within depth BFS steps. If any of those states appear later in the original path (beyond current position + BFS steps taken), the algorithm "jumps" to the farthest such match, replacing the skipped segment with the shorter BFS route. States are indexed in a hash map supporting duplicate entries to catch the farthest possible jumps in paths with repeated states.

Usage

```
short_path_bfs(path, start_state, k, depth = 5L)
```

Arguments

path	Character vector of operations ("1"/"2"/"3" or "L"/"R"/"X")
start_state	Integer vector, the starting permutation state
k	Integer, parameter for reverse_prefix operation
depth	Integer, BFS exploration depth (default 5)

Value

List with path (shortened), original_length, new_length, savings

short_position	<i>Simplify Operation Path</i>
----------------	--------------------------------

Description

Removes redundant operations from a path: cancels inverse pairs ("1"+"2", "3"+"3"), reduces chains of shifts modulo n, and simplifies blocks between reverses.

Usage

```
short_position(allowed_positions, n)
```

Arguments

allowed_positions	Character vector of operations to simplify
n	Integer, size of the permutation ring (used for modular reduction)

Value

Character vector of simplified operations

Examples

```
short_position(c("1", "2"), n = 5)
short_position(c("3", "3"), n = 5)
short_position(c("1", "1", "1", "1", "1"), n = 5)
```

 sparse_bfs

Sparse BFS with Look-ahead and Hybrid Selection

Description

Sparse BFS with Look-ahead and Hybrid Selection

Usage

```
sparse_bfs(start_state, k, n_hubs = 7L, n_random = 3L, max_levels = 1000L)
```

Arguments

start_state	Integer vector — starting permutation
k	Integer — parameter for reverse_prefix operation
n_hubs	Number of top-degree candidates to keep per level (exploitation)
n_random	Number of random candidates to keep per level (exploration)
max_levels	Maximum BFS depth (default 1000)

Value

data.frame with columns: parent_key, child_key, operation, level

 state_store_query

Query a State Store

Description

Accessors for a StateStore created by [create_state_store](#). The implementations come from the C++ layer; these blocks document and export them.

Usage

```
state_store_size(xp)
state_store_perm_length(xp)
state_store_unique_count(xp)
state_store_indices_for_cycle(xp, target_cycle)
```

Arguments

xp	External pointer to StateStore
target_cycle	Integer, cycle number to look up

Details

state_store_size Number of states currently stored.

state_store_perm_length Length of each permutation state.

state_store_unique_count Number of distinct states.

state_store_indices_for_cycle Row indices belonging to a given cycle.

The functions themselves are generated into RcppExports.R; these @export tags are what put them in the NAMESPACE.

Value

Integer, or an integer vector for state_store_indices_for_cycle

Examples

```
store <- create_state_store(6L)
state_store_size(store)
state_store_perm_length(store)
```

store_add_from_df	<i>Add States to Store from Data Frame</i>
-------------------	--

Description

Converts a data.frame/data.table of states (as returned by analyze_top_combinations) into the compact C++ store.

Usage

```
store_add_from_df(store, df, cycle_val)
```

Arguments

store	External pointer to StateStore
df	Data frame with V1..Vn columns plus metadata
cycle_val	Integer, cycle number to assign

Value

Number of rows added (invisible)

store_analyze_combos	<i>Analyze Combinations Directly into Store</i>
----------------------	---

Description

C++ implementation that runs full cycle expansion for each combination and writes states + coordinates directly into the StateStore, bypassing data.frame creation entirely.

Usage

```
store_analyze_combos(store, top_combos, start_state, k, cycle_val)
```

Arguments

store	External pointer to StateStore
top_combos	Data frame with combination column
start_state	Integer vector, the initial permutation state
k	Integer, parameter for reverse operations
cycle_val	Integer, cycle number to assign

Value

Number of states added (invisible)

store_analyze_combos_gpu	<i>Analyze Combinations into Store Using GPU Batch Operations</i>
--------------------------	---

Description

GPU-accelerated version of store_analyze_combos. Processes all combinations in parallel, step by step, grouping by operation type (L/R/X) for GPU matrix multiplication. Falls back to CPU if GPU is unavailable.

Usage

```
store_analyze_combos_gpu(store, top_combos, start_state, k, cycle_val)
```

Arguments

store	External pointer to StateStore
top_combos	Data frame with combination column
start_state	Integer vector, the initial permutation state
k	Integer, parameter for reverse operations
cycle_val	Integer, cycle number to assign

Value

Number of states added (invisible)

store_clear	<i>Drop All States From a Store</i>
-------------	-------------------------------------

Description

Removes every stored state and rebuilds no indices, keeping the allocated capacity so the store can be refilled without reallocating. Frees memory immediately, unlike replacing the store with a fresh one (which defers release to R's garbage collector and can hold two stores at once).

Usage

store_clear(store)

Arguments

store External pointer to StateStore

store_clear_opd	<i>Clear All OPD Filters</i>
-----------------	------------------------------

Description

Clear All OPD Filters

Usage

store_clear_opd(store)

Arguments

store External pointer to StateStore

store_collect_ops	<i>Collect Operations Leading to a State Within One Cycle</i>
-------------------	---

Description

Returns the operation sequence from the start of target_combo up to (not including) end_step. Used to capture the path segment reaching a bridge while that cycle’s states are still in the store, so the segment survives [store_clear](#).

Usage

store_collect_ops(store, target_cycle, target_combo, end_step)

Arguments

store	External pointer to StateStore
target_cycle	Integer, cycle the state belongs to
target_combo	Integer, combo_number of the state
end_step	Integer, step of the state; NA yields no operations (the state is the combo’s starting point)

Value

Character vector of operations

store_combos_for_state	<i>Find Combo Numbers Containing a State in a Cycle</i>
------------------------	---

Description

Find Combo Numbers Containing a State in a Cycle

Usage

store_combos_for_state(store, state, target_cycle)

Arguments

store	External pointer to StateStore
state	Integer vector
target_cycle	Integer

Value

Integer vector of combo_numbers

store_filter_middle	<i>Filter Middle States for a Cycle</i>
---------------------	---

Description

Returns 0-based indices of states in the given cycle, excluding the first skip_first and last skip_last steps per combo.

Usage

```
store_filter_middle(store, target_cycle, skip_first = 5L, skip_last = 5L)
```

Arguments

store	External pointer to StateStore
target_cycle	Integer
skip_first	Integer (default 5)
skip_last	Integer (default 5)

Value

Integer vector of 0-based indices

store_find_best_match	<i>Find Best Match by Manhattan Distance</i>
-----------------------	--

Description

Find Best Match by Manhattan Distance

Usage

```
store_find_best_match(store, target, candidate_indices = integer(0))
```

Arguments

store	External pointer to StateStore
target	Integer vector, target state
candidate_indices	Integer vector of 0-based indices to search (empty = search all)

Value

Integer, 0-based index of best match

store_find_intersections	<i>Find Intersections Between Two Stores</i>
--------------------------	--

Description

Returns state keys present in both stores. $O(\min(N,M))$ via hash lookup.

Usage

```
store_find_intersections(store_a, store_b)
```

Arguments

- store_a External pointer to StateStore
- store_b External pointer to StateStore

Value

Character vector of common state keys

store_get_meta	<i>Get Metadata for a State</i>
----------------	---------------------------------

Description

Retrieves metadata (step, combo_number, cycle, operation, coordinates) for a single state by 0-based index.

Usage

```
store_get_meta(store, idx)
```

Arguments

- store External pointer to StateStore
- idx Integer, 0-based index

Value

Named list

store_get_state	<i>Get State from Store</i>
-----------------	-----------------------------

Description

Retrieves a single permutation state by 0-based index.

Usage

```
store_get_state(store, idx)
```

Arguments

- | | |
|-------|--------------------------------|
| store | External pointer to StateStore |
| idx | Integer, 0-based index |

Value

Integer vector of length perm_length

store_lookup	<i>Lookup State Indices by State Vector</i>
--------------	---

Description

Lookup State Indices by State Vector

Usage

```
store_lookup(store, state)
```

Arguments

- | | |
|-------|--------------------------------|
| store | External pointer to StateStore |
| state | Integer vector |

Value

Integer vector of 0-based indices

store_reconstruct_path	<i>Reconstruct Path from Store</i>
------------------------	------------------------------------

Description

Traces back through cycle chain using bridge states to build the correct operation sequence. Each bridge state defines which combo path to follow in each cycle.

Usage

```
store_reconstruct_path(  
    store,  
    bridge_states,  
    target_state,  
    target_cycle,  
    target_combo  
)
```

Arguments

store	External pointer to StateStore
bridge_states	List of bridge state entries, each with \$state (integer vector). Element 1 = root (cycle 0), element i+1 = bridge at cycle i.
target_state	Integer vector
target_cycle	Integer
target_combo	Integer

Value

Character vector of operations, or NULL

store_set_opd	<i>Set OPD Combo Filter for a Cycle</i>
---------------	---

Description

Restricts indices_for_cycle and filter_middle_indices to only return states from the specified combo_numbers for the given cycle.

Usage

```
store_set_opd(store, target_cycle, combos)
```

Arguments

store	External pointer to StateStore
target_cycle	Integer, cycle number to filter
combos	Integer vector of allowed combo_numbers

store_to_dataframe	<i>Convert Store to Data Frame</i>
--------------------	------------------------------------

Description

For debugging and backward compatibility. Converts the entire store contents to a data.frame with V1..Vn + metadata columns.

Usage

```
store_to_dataframe(store)
```

Arguments

store	External pointer to StateStore
-------	--------------------------------

Value

data.frame

validate_and_simplify_path	<i>Validate and Simplify a Path</i>
----------------------------	-------------------------------------

Description

Verifies that a candidate path correctly transforms start_state into final_state, then attempts to simplify it. Returns the simplified path if it remains valid, otherwise the original.

Usage

```
validate_and_simplify_path(path_candidate, start_state, final_state, k)
```

Arguments

path_candidate	Character vector of operations
start_state	Integer vector, start state
final_state	Integer vector, target state
k	Integer, parameter for reverse operations

Value

List with components:

<code>valid</code>	Logical, whether the path is valid
<code>path</code>	Simplified or original path, or NULL if invalid

Examples

```
res <- validate_and_simplify_path(c("1", "3"), 1:5, c(5, 2, 3, 4, 1), k = 2)
res$valid
```

Index

analyze_top_combinations, [3](#)
apply_operations, [4](#)
apply_operations_batch_gpu, [5](#)

bidirectional_bfs, [6](#)
breakpoint_distance, [6](#)
build_permutation_matrix, [7](#)

calculate_angular_distance_z, [8](#)
calculate_differences, [8](#)
calculate_midpoint_z, [9](#)
cayley_bfs_full, [10](#), [14](#)
cayley_distance (distance_methods), [20](#)
cayley_distance_methods
 (distance_methods), [20](#)
cayley_gpu_available, [11](#)
cayley_gpu_free, [11](#)
cayley_gpu_init, [12](#)
cayley_gpu_status, [12](#)
cayley_graph_diameter, [10](#), [13](#)
check_duplicates, [14](#)
compose_permutation_matrix, [15](#)
convert_digits, [15](#)
convert_LRX_to_celestial, [10](#), [16](#)
convex_hull_3d, [17](#), [21](#), [22](#)
create_state_store, [18](#), [48](#)
cycle_shortcut, [18](#), [40](#)

distance_methods, [20](#), [23](#)

enclosing_hull_3d, [21](#)

find_best_match_human, [21](#), [22](#)
find_best_random_combinations, [19](#), [23](#),
 [28](#)
find_closest_to_coords, [25](#)
find_combination_in_states, [25](#)
find_path_bfs, [26](#)
find_path_iterative, [18](#), [20](#), [21](#), [27](#), [34](#)
generate_state, [29](#)

generate_unique_states_df, [30](#)
get_reachable_states, [30](#)
get_reachable_states_light, [31](#)

human_algorithm, [32](#), [33–36](#), [42](#)
human_algorithm_to, [20](#), [33](#)
human_phase1_navigate, [34](#)
human_phase1_rank, [21](#), [23](#), [34](#), [35](#), [35](#), [42](#)

invert_path, [36](#)

landmark_states, [37](#)

manhattan_distance, [38](#)
manhattan_distance_matrix_gpu, [39](#)

openmp_threads, [40](#)

reconstruct_bfs_path, [40](#)
reverse_prefix, [41](#)
reverse_prefix_simple, [41](#)
run_length, [36](#), [42](#)

save_bridge_states, [43](#)
select_unique, [43](#)
shift_left, [44](#)
shift_left_simple, [45](#)
shift_right, [45](#)
shift_right_simple, [46](#)
short_path_bfs, [19](#), [20](#), [46](#)
short_position, [47](#)
sparse_bfs, [10](#), [48](#)
state_store_indices_for_cycle
 (state_store_query), [48](#)
state_store_perm_length
 (state_store_query), [48](#)
state_store_query, [48](#)
state_store_size (state_store_query), [48](#)
state_store_unique_count
 (state_store_query), [48](#)
store_add_from_df, [49](#)

store_analyze_combos, [50](#)
store_analyze_combos_gpu, [50](#)
store_clear, [51](#), [52](#)
store_clear_opd, [51](#)
store_collect_ops, [52](#)
store_combos_for_state, [52](#)
store_filter_middle, [53](#)
store_find_best_match, [53](#)
store_find_intersections, [54](#)
store_get_meta, [54](#)
store_get_state, [55](#)
store_lookup, [55](#)
store_reconstruct_path, [56](#)
store_set_opd, [56](#)
store_to_dataframe, [57](#)

validate_and_simplify_path, [57](#)