Package: cayleyR (via r-universe)

May 29, 2026

Type Package

Title Cayley Graph Analysis for Permutation Puzzles

Version 0.2.3

Description Implements algorithms for analyzing Cayley graphs of permutation groups for the TopSpin puzzle. Provides methods for cycle detection, state space exploration, bidirectional BFS pathfinding, and finding optimal operation sequences in permutation groups generated by shift and reverse operations. Includes C++ implementations of core operations via 'Rcpp' for performance, including a depth-limited BFS path shortener implemented entirely in C++.

License MIT + file LICENSE

Encoding UTF-8

LazyData true

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.3

Imports Rcpp

LinkingTo Rcpp

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

Config/testthat/edition 3

URL https://github.com/Zabis13/cayleyR

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

Repository https://zabis13.r-universe.dev

Date/Publication 2026-04-09 20:51:58 UTC

RemoteUrl https://github.com/zabis13/cayleyr

RemoteRef HEAD

RemoteSha 8d2d05b926076ad0db632e1e84696c0b08f74f6f

Contents

`analyze_top_combinations`
Analyze Top Operation Combinations

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 *Apply Sequence of Operations*

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.

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.

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 *Bidirectional BFS Shortest Path*

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 *Breakpoint Distance Between Two States*

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)
```

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

result1	List with component z (complex number)
result2	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_gpu_available *Check if GPU acceleration is available*

Description

Checks whether ggmlR is installed and Vulkan GPU is present.

Usage

```
cayley_gpu_available()
```

Value

Logical

Examples

```
cayley_gpu_available()
```

cayley_gpu_free *Free GPU backend resources*

Description

Free GPU backend resources

Usage

```
cayley_gpu_free()
```

Value

Invisible NULL

cayley_gpu_init *Initialize GPU backend*

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 *Get GPU status information*

Description

Get GPU status information

Usage

```
cayley_gpu_status()
```

Value

List with availability, device info, and backend status

Examples

```
cayley_gpu_status()
```

check_duplicates *Find Duplicate States Between Two Tables*

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)
```

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

s	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
```

---

create_state_store *Create a New State Store*

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)
```

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

Value

Data frame with columns:

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

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` *Find Path via BFS Highways*

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,
  distance_method = "manhattan",
  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)
`distance_method`	Character, "manhattan" or "breakpoints" (default "manhattan")
`verbose`	Logical, print progress (default TRUE)
`...`	Additional arguments passed to find_path_iterative

Value

List with path, found, cycles, bfs_info

`find_path_iterative` *Iterative Path Finder Between Permutation States*

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,
  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)
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 *Generate Reachable Random State*

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")
```

invert_path *Invert a Path of Operations*

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"))
```

---

manhattan_distance *Manhattan Distance Between Two States*

Description

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

Usage

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

Arguments

start_state	Integer vector, first state
target_state	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 x 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

states1	Numeric matrix (N1 x n), first set of states
states2	Numeric matrix (N2 x n), second set of states
batch_size	Integer, number of states2 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 x 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)
}
```

reconstruct_bfs_path *Reconstruct path from sparse BFS result*

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 *Reverse First k Elements (with Coordinates)*

Description

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

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 *Reverse First k Elements (Simple)*

Description

Simple prefix reversal without coordinate tracking.

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)
```

save_bridge_states *Save Bridge States to CSV*

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 *Select Unique States by V-columns*

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 *Shift State Left (with Coordinates)*

Description

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

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` *Shift State Left (Simple)*

Description

Simple cyclic left shift without coordinate tracking.

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` *Shift State Right (with Coordinates)*

Description

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

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 *Shift State Right (Simple)*

Description

Simple cyclic right shift without coordinate tracking.

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 *Shorten Path via Depth-Limited BFS Hopping*

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 *Simplify Operation Path*

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_size *Get State Store Size*

Description

Returns the number of states currently stored in the StateStore.

Arguments

xp	External pointer to StateStore

Value

Integer, number of stored states

Examples

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

store_add_from_df *Add States to Store from Data Frame*

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 *Analyze Combinations Directly into Store*

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

Analyze Combinations into Store Using GPU Batch Operations

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_opd

Clear All OPD Filters

Description

Clear All OPD Filters

Usage

```
store_clear_opd(store)
```

Arguments

`store`	External pointer to StateStore

store_combos_for_state *Find Combo Numbers Containing a State in a Cycle*

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 *Filter Middle States for a Cycle*

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 *Find Best Match by Manhattan Distance*

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 *Find Intersections Between Two Stores*

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 *Get Metadata for a State*

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 *Get State from Store*

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 *Lookup State Indices by State Vector*

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 *Reconstruct Path from Store*

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` *Set OPD Combo Filter for a Cycle*

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` *Convert Store to Data Frame*

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 *Validate and Simplify a Path*

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:

valid	Logical, whether the path is valid
path	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