

# Package ‘cayleyR’

May 8, 2026

**Type** Package

**Title** Cayley Graph Analysis for Permutation Puzzles

**Version** 0.2.1

**Description** Implements algorithms for analyzing Cayley graphs of permutation groups, with a focus on the TopSpin puzzle and similar permutation-based combinatorial puzzles. 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. Optional GPU acceleration via 'ggmlR' Vulkan backend for batch distance calculations and parallel state transformations.

**License** MIT + file LICENSE

**Encoding** UTF-8

**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>

**NeedsCompilation** yes

**Author** Yuri Baramykov [aut, cre]

**Maintainer** Yuri Baramykov <lbsbmsu@mail.ru>

**Repository** CRAN

**Date/Publication** 2026-03-01 13:30:17 UTC

## Contents

analyze_top_combinations . . . . .	3
apply_operations . . . . .	3
apply_operations_batch_gpu . . . . .	4
bidirectional_bfs . . . . .	5
breakpoint_distance . . . . .	5
calculate_angular_distance_z . . . . .	6
calculate_differences . . . . .	7
calculate_midpoint_z . . . . .	7
cayley_gpu_available . . . . .	8
cayley_gpu_free . . . . .	8
cayley_gpu_init . . . . .	9
cayley_gpu_status . . . . .	9
check_duplicates . . . . .	10
convert_digits . . . . .	10
convert_LRX_to_celestial . . . . .	11
find_best_random_combinations . . . . .	12
find_closest_to_coords . . . . .	13
find_combination_in_states . . . . .	14
find_path_bfs . . . . .	14
find_path_iterative . . . . .	15
generate_state . . . . .	17
generate_unique_states_df . . . . .	18
get_reachable_states . . . . .	18
get_reachable_states_light . . . . .	19
invert_path . . . . .	20
manhattan_distance . . . . .	21
manhattan_distance_matrix_gpu . . . . .	21
reconstruct_bfs_path . . . . .	22
reverse_prefix . . . . .	23
reverse_prefix_simple . . . . .	23
save_bridge_states . . . . .	24
select_unique . . . . .	24
shift_left . . . . .	25
shift_left_simple . . . . .	26
shift_right . . . . .	26
shift_right_simple . . . . .	27
short_path_bfs . . . . .	27
short_position . . . . .	28
sparse_bfs . . . . .	28
validate_and_simplify_path . . . . .	29

---

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	<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()
```

---

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)
```

---

convert_digits	<i>Convert String to Integer Vector of Digits</i>
----------------	---

---

**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

```

---

 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 longest 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
)
```

## 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 (sorted by cycle length)
start_state	Integer vector, initial permutation state
k	Integer, parameter for reverse operations

## 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

**Examples**

```
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  
)  
print(best)
```

---

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",
  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".
verbose	Logical, if TRUE prints progress messages (default TRUE)

**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"))
```

### 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"))

### 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	<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**

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	<i>Compute Pairwise Manhattan Distance Matrix on GPU</i>
-------------------------------	--

---

**Description**

Computes all pairwise Manhattan distances between two sets of states. Returns an  $N_1 \times N_2$  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 ( $N_1 \times n$ ), first set of states
states2	Numeric matrix ( $N_2 \times n$ ), 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	<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).

**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.

**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	<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).

**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.

**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).

**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.

**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 Greedy BFS Hopping</i>
----------------	--

---

**Description**

Shorten Path via Greedy BFS Hopping

**Usage**

```
short_path_bfs(path, start_state, k, n_hits = 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
n_hits	Integer, number of path points to find in BFS cloud (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	<i>Sparse BFS with Look-ahead and Hybrid Selection</i>
------------	--

---

**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

---

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

analyze\_top\_combinations, [3](#)  
apply\_operations, [3](#)  
apply\_operations\_batch\_gpu, [4](#)  
  
bidirectional\_bfs, [5](#)  
breakpoint\_distance, [5](#)  
  
calculate\_angular\_distance\_z, [6](#)  
calculate\_differences, [7](#)  
calculate\_midpoint\_z, [7](#)  
cayley\_gpu\_available, [8](#)  
cayley\_gpu\_free, [8](#)  
cayley\_gpu\_init, [9](#)  
cayley\_gpu\_status, [9](#)  
check\_duplicates, [10](#)  
convert\_digits, [10](#)  
convert\_LRX\_to\_celestial, [11](#)  
  
find\_best\_random\_combinations, [12](#)  
find\_closest\_to\_coords, [13](#)  
find\_combination\_in\_states, [14](#)  
find\_path\_bfs, [14](#)  
find\_path\_iterative, [15](#)  
  
generate\_state, [17](#)  
generate\_unique\_states\_df, [18](#)  
get\_reachable\_states, [18](#)  
get\_reachable\_states\_light, [19](#)  
  
invert\_path, [20](#)  
  
manhattan\_distance, [21](#)  
manhattan\_distance\_matrix\_gpu, [21](#)  
  
reconstruct\_bfs\_path, [22](#)  
reverse\_prefix, [23](#)  
reverse\_prefix\_simple, [23](#)  
  
save\_bridge\_states, [24](#)  
select\_unique, [24](#)  
shift\_left, [25](#)  
  
shift\_left\_simple, [26](#)  
shift\_right, [26](#)  
shift\_right\_simple, [27](#)  
short\_path\_bfs, [27](#)  
short\_position, [28](#)  
sparse\_bfs, [28](#)  
  
validate\_and\_simplify\_path, [29](#)