Package: cayleyR (via r-universe)

October 20, 2026

Type Package

Title Cayley Graph Analysis for Permutation Puzzles

Version 0.3.0

Description Analyzes Cayley graphs of permutation puzzles. Any puzzle whose moves permute a fixed set of positions can be declared as a permutation group, and the same tools then work on it: cycle detection, state space exploration and path search. Two puzzles are built in: TopSpin and Rubik's cubes of any size. The 3x3x3 is solved by four human methods and by a two-phase search after Herbert Kociemba; the 4x4x4 is solved by reduction. The Iterative Cycle Intersection (ICI) method is described in Yuri Baramykov (2026) <doi:10.48550/arXiv.2607.13219>.

License MIT + file LICENSE

Encoding UTF-8

RoxygenNote 7.3.3

Imports Rcpp, utils

LinkingTo Rcpp

Suggests testthat (>= 3.0.0), ggmlR, parallel, data.table, jsonlite, 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>), Lucas Garron [ctb, cph] (twips search machinery in src/kociemba_core.h, src/kociemba4.h), cubing.js authors [cph] (twips, https://github.com/cubing/twips)

Maintainer Yuri Baramykov <lbsbmsu@mail.ru>

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

Date/Publication 2026-10-19 12:20:08 UTC

RemoteUrl https://github.com/cran/cayleyR

RemoteRef HEAD

RemoteSha eec42c4ca1f01cf4b80328faf6f2aebac202aff3

Contents

analyze_top_combinations . . . 5
apply_operations . . . 6
apply_operations_batch_gpu . . . 7
bidirectional_bfs . . . 8
breakpoint_distance . . . 9
build_permutation_matrix . . . 10
calculate_angular_distance_z . . . 10
calculate_differences . . . 11
calculate_midpoint_z . . . 12
cayley_bfs_full . . . 12
cayley_gpu_available . . . 14
cayley_gpu_free . . . 14
cayley_gpu_init . . . 15
cayley_gpu_status . . . 15
cayley_graph_diameter . . . 16
check_duplicates . . . 17
compose_permutation_matrix . . . 18
convert_digits . . . 18
convert_LRX_to_celestial . . . 19
convex_hull_3d . . . 20
create_state_store . . . 21
cube_adi . . . 22
cube_adi_astar . . . 23
cube_adi_load . . . 24
cube_adi_model . . . 25
cube_adi_save . . . 26
cube_adi_solve . . . 27
cube_adi_train . . . 27
cube_alg_table . . . 28
cube_apply_word . . . 29
cube_build_lslice . . . 30
cube_central_moves . . . 31
cube_centre_counts . . . 32
cube_centre_cycles . . . 33
cube_centre_map_family . . . 34
cube_centre_positions . . . 35
cube_centre_shots . . . 36

cube_centre_structure . . . 37
cube_colours . . . 38
cube_edge_algs . . . 39
cube_edge_counts . . . 40
cube_edge_map . . . 41
cube_edge_map_family . . . 42
cube_edge_structure . . . 43
cube_empty_u_slice . . . 44
cube_expand_alg . . . 45
cube_expand_move . . . 46
cube_group . . . 48
cube_identity . . . 49
cube_is_colour_solved . . . 50
cube_is_reduced . . . 51
cube_kociemba . . . 52
cube_kociemba_init . . . 54
cube_kociemba_report . . . 55
cube_kociemba4 . . . 55
cube_kociemba4_reduce . . . 57
cube_kociemba4_report . . . 59
cube_layer_move . . . 59
cube_lslice_cells . . . 60
cube_move_names . . . 61
cube_moves . . . 62
cube_nnn . . . 63
cube_orbits . . . 64
cube_orbits_family . . . 64
cube_pair_edges . . . 65
cube_piece_layout . . . 66
cube_pieces . . . 67
cube_pieces_home . . . 68
cube_predicates . . . 69
cube_progress . . . 69
cube_read_state . . . 70
cube_reduce_best . . . 71
cube_reduce_cpp . . . 72
cube_santa . . . 73
cube_santa_group . . . 74
cube_santa_move_names . . . 75
cube_santa_moves . . . 75
cube_santa_path . . . 76
cube_santa_perm . . . 77
cube_santa_state . . . 78
cube_slice_cells_cpp . . . 79
cube_slice_map . . . 80
cube_solve_centres . . . 81
cube_solve_cfop . . . 82
cube_solve_lbl . . . 83

cube_solve_m2 . . . 84
cube_solve_old_pochmann . . . 85
cube_solve4 . . . 86
cube_solve4_cascade . . . 89
cube_wide_turn . . . 92
cube_word_order . . . 93
cycle_shortcut . . . 94
distance_methods . . . 96
enclosing_hull_3d . . . 97
find_best_match_cube4_model . . . 98
find_best_match_human . . . 99
find_best_random_combinations . . . 100
find_closest_to_coords . . . 102
find_combination_in_states . . . 103
find_path_bfs . . . 103
find_path_iterative . . . 105
generate_state . . . 107
generate_unique_states_df . . . 108
generate_walk . . . 109
get_reachable_states . . . 111
get_reachable_states_light . . . 112
group_apply . . . 113
group_compose . . . 114
group_identity . . . 114
group_inverse_seq . . . 115
group_move_index . . . 116
group_moves . . . 117
group_order . . . 117
group_parse_word . . . 118
human_algorithm . . . 119
human_algorithm_to . . . 120
human_phase1_navigate . . . 121
human_phase1_rank . . . 122
invert_path . . . 123
is_perm_group . . . 124
kociemba4_phases . . . 124
landmark_states . . . 126
manhattan_distance . . . 128
manhattan_distance_matrix_gpu . . . 128
openmp_threads . . . 129
perm_group . . . 130
perm_group_intro . . . 131
print.perm_group . . . 131
reconstruct_bfs_path . . . 132
reverse_prefix . . . 133
reverse_prefix_simple . . . 133
run_length . . . 134
save_bridge_states . . . 135

select_unique . . . 135
shift_left . . . 136
shift_left_simple . . . 137
shift_right . . . 137
shift_right_simple . . . 138
short_path_bfs . . . 138
short_position . . . 139
sparse_bfs . . . 140
state_store_query . . . 141
store_add_from_df . . . 142
store_analyze_combos . . . 142
store_analyze_combos_gpu . . . 143
store_clear . . . 144
store_clear_opd . . . 144
store_collect_ops . . . 145
store_combos_for_state . . . 145
store_filter_middle . . . 146
store_find_best_match . . . 146
store_find_intersections . . . 147
store_get_meta . . . 147
store_get_state . . . 148
store_group . . . 148
store_lookup . . . 149
store_reconstruct_path . . . 149
store_set_opd . . . 150
store_to_dataframe . . . 150
topspin_group . . . 151
validate_and_simplify_path . . . 152

Index **154**

`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.

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` *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 = NULL,
  k = NULL,
  group = NULL
)
```

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`	Allowed operations, naming a subset of the group's alphabet (default: all of it)
`k`	Integer, parameter for reverse operations. Ignored when `group` is given.
`group`	A `perm_group`. When `NULL` (default) the arguments describe TopSpin.

Details

The backward half of the search walks moves in reverse, so the path it returns may name moves outside `moves`: searching with left shifts alone still reports the right shift that undoes one. Inverses are taken from the group's whole alphabet for that reason, and a move the group cannot undo at all is an error rather than a silently wrong path.

Value

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

See Also

`perm_group`

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

# the same search on a cube: undo a three-move scramble
g <- cube_group(3)
s <- group_apply(g, group_identity(g), "R U F")
bidirectional_bfs(54, s, group_identity(g), max_level = 4, group = g)
```

---

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)
```

`build_permutation_matrix` *Build permutation matrix for a single operation*

Description

Creates an n x n permutation matrix (column-major, F32) that represents a single operation. When multiplied: new_state = 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

`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_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 = NULL, moves = NULL, group = NULL)
```

Arguments

start_state	Integer vector, the state to explore from
k	Integer, parameter for the reverse-prefix operation. Ignored when group is given, since a group already knows what its own moves do.
moves	Allowed operations, naming a subset of the group's alphabet (default: all of it). For TopSpin these are c("L","R","X") or c("1","2","3") as before.
group	A perm_group. When NULL (default) the arguments describe TopSpin and the group is built from k and moves.

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.

The search runs over a perm_group, so the same sweep enumerates a TopSpin ring or a Rubik's cube. Passing k without a group means TopSpin, exactly as before.

Value

Data frame with one row per reachable state:

state_str	State as an underscore-separated key
dist	Graph distance from start_state
nL, nR, nX	Operation counts along the BFS shortest path. Zero for groups other than TopSpin, whose alphabet has no L/R/X to count
theta, phi, omega	Celestial coordinates derived from those counts

See Also

cayley_graph_diameter, sparse_bfs, perm_group

Examples

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

# the same sweep over a cube subgroup: the middle slice, on its own
g <- cube_group(3, moves = c("M", "M'"))
d2 <- cayley_bfs_full(group_identity(g), group = g)
nrow(d2)
```

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()
```

`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 = NULL,
  moves = NULL,
  group = NULL,
  method = c("all_pairs", "from_start"),
  max_pairs = Inf,
  verbose = FALSE
)
```

Arguments

`start_state`	Integer vector, the state to explore from
`k`	Integer, parameter for the reverse-prefix operation. Ignored when `group` is given.
`moves`	Allowed operations, naming a subset of the group's alphabet (default: all of it)
`group`	A `perm_group`. When `NULL` (default) the arguments describe TopSpin, as they always have.
`method`	Either "all_pairs" (default, exact) or "from_start" (single BFS, exact only for vertex-transitive graphs)
`max_pairs`	Numeric, maximum number of pairs to materialise in `pairs_df` (default `Inf`). `n_pairs` always reports the honest total.
`verbose`	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:

diameter	Integer, the graph diameter (or start eccentricity)
n_vertices	Number of reachable states
n_pairs	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)

# God's number for a small cube subgroup: the middle slice, on its own
g <- cube_group(3, moves = c("M", "M'"))
cayley_graph_diameter(group_identity(g), group = g,
                      method = "from_start")$diameter
```

---

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)
```

---

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

`operations`	Character vector of operation codes
`n`	Integer, state length
`k`	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

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
```

`convex_hull_3d` *Convex Hull of a 3-D Point Cloud*

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` *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, group = NULL)
```

Arguments

`perm_length`	Integer, length of each permutation state
`init_capacity`	Integer, initial capacity (default 10000)
`group`	A `perm_group`, or `NULL` (default)

Details

A store may be told which `perm_group` its states belong to. It needs this only to spell its own `operation` column: the codes it keeps are indices into the group's alphabet, and nothing but the group knows whether index 2 means a shift or a face turn. Without one, the legacy TopSpin spelling "1"/"2"/"3" is used.

Value

External pointer to StateStore (XPtr), with the group attached as an attribute so it stays alive as long as the store does

Examples

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

# a store that speaks cube
g <- cube_group(3)
cs <- create_state_store(54L, group = g)
state_store_perm_length(cs)
```

cube_adi *Autodidactic Iteration on a Permutation Puzzle*

Description

Training a network to solve a cube without ever being shown a solution.

Details

The difficulty with learning a distance-to-solved is that nobody knows what it is. Labelling a state with the length of the scramble that produced it is the obvious thing to do and it is wrong: the scramble is one path to that state, not the shortest, so the label is an upper bound that gets looser the deeper the scramble goes, and a network trained on it learns the looseness along with the distance.

Autodidactic iteration sidesteps this. A state's target is not its scramble length but $\min_a(1 + v(child_a))$ — one move, plus whatever the network currently believes about the best of its children. That looks circular, and would be, except that a child which is already solved counts as zero no matter what the network says. States one move from solved therefore get an exact target immediately, their neighbours get an almost exact one from them, and correctness spreads outward as training goes on. This is why scrambles are drawn uniformly over depth and the loss carries no weighting: the ordering is in the structure of the targets, not the sampling.

Two networks

Value and policy are separate models here, not two heads on one trunk. `ggml_fit` trains a single output, so a shared trunk would need a multi-output loss that ggmlR does not yet have. The cost is a second forward pass per move at solve time; the benefit is that each network trains under the loss that suits it – mean squared error for a distance, cross-entropy for a choice of move – with no changes to ggmlR.

The frozen network

Targets are computed with a copy of the value network held fixed for the whole batch. Without it the targets move as the network moves and training chases itself; the copy is refreshed only once the running loss drops below `loss_thresh`, which is DeepCubeA's criterion and steadier than refreshing on a fixed schedule.

See Also

`cube_adi_train`, `cube_adi_solve`

cube_adi_astar *Batch Weighted A* With a Learned Heuristic*

Description

The search half of DeepCubeA. cube_adi_train learns a distance to solved; this walks the graph using that distance as its heuristic, and it is what turns an estimator into a solver.

Usage

```
cube_adi_astar(
  net,
  state,
  weight = 0.6,
  batch = 1000L,
  max_nodes = 1e+06,
  batch_size = 256L,
  verbose = FALSE
)
```

Arguments

net	A trained cube_adi_net from cube_adi_train
state	Integer vector, the scrambled state
weight	Weight λ on the cost so far. 0 is greedy best-first, 1 is A*.
batch	Nodes expanded per iteration
max_nodes	Give up after this many nodes have been expanded
batch_size	Batch size the network was compiled at; children are padded out to a multiple of it, as in cube_adi_solve
verbose	TRUE to report each iteration

Value

List with solved, path (move names), length, nodes (expanded) and iterations

Why not simply walk downhill

cube_adi_solve takes the best-looking child at every step and never reconsiders. That is enough near the goal, where the network is exact, and fails past it: a state whose true distance the network underrates draws the walk in, every child then looks worse than the parent, and the walk is stuck with no way back. A search keeps the alternatives it passed over on a list and returns to them when the branch it chose stops paying, so a wrong estimate costs time rather than the solve.

The weight

Nodes are ordered by $f = \lambda g + h$ — g moves spent to reach the node, h the network's guess at what remains. $\lambda = 1$ is textbook A*, which returns the shortest path the heuristic admits but expands far more nodes; $\lambda = 0$ ignores the cost so far and is pure greedy best-first, fast and prone to wandering. DeepCubeA runs around 0.6, buying much shorter runtimes for slightly longer solutions, and that is the default here. Note that the learned heuristic is not admissible in any case, so even $\lambda = 1$ carries no optimality guarantee — the weight trades length against time, it does not switch a guarantee on.

Why a batch

The expensive part of an iteration is not the queue, it is the forward pass. Scoring one node's children is a batch of `n_moves`, which leaves a GPU idle; so each iteration pops the `batch` best nodes at once, expands all of them, and scores every child in a single call. The cost of an iteration is then nearly flat in `batch`, and a wider batch means fewer of them. It also loosens the ordering — the 1000th-best node is expanded alongside the best — which is the same trade the weight makes, and the reason a very wide batch can lengthen solutions.

See Also

`cube_adi_solve` for the greedy walk

`cube_adi_load` *Load a Trained ADI Network*

Description

Inverse of `cube_adi_save`. The group is rebuilt from the saved permutations, so the result is usable by `cube_adi_solve` and `cube_adi_train` without any of the code that first created it.

Usage

```
cube_adi_load(path, backend = "auto")
```

Arguments

`path`	Directory written by `cube_adi_save`
`backend`	Backend for the loaded models: `"cpu"`, `"vulkan"` or `"auto"`

Value

A `cube_adi_net`

See Also

`cube_adi_save`

cube_adi_model *Build the Value and Policy Networks*

Description

Two models over the same input: 96 sticker positions read as tokens through a shared-in-shape, separate-in-weights embedding, then a stack of dense layers. The value network ends in one linear unit (a distance), the policy network in one softmax unit per move.

Usage

```
cube_adi_model(
  group,
  embed_dim = 32L,
  hidden = c(1024L, 512L),
  arch = c("mlp", "resnet", "lstm", "gru", "transformer"),
  rnn_units = 256L,
  n_blocks = 4L,
  d_model = 64L,
  n_heads = 4L,
  ff_dim = NULL,
  encoding = c("sticker", "piece"),
  backend = "auto"
)
```

Arguments

group	A perm_group, used for its state length and move count
embed_dim	Width of the token embedding (arch = "mlp" only)
hidden	Integer vector of hidden layer widths
arch	"mlp" (default) reads the state as a set of tokens through an embedding and puts a dense stack on top; "resnet" is the same input under residual blocks, which is the shape DeepCubeA uses. "lstm" and "gru" instead read the state as a sequence, one position at a time. A cube state is a set, not a sequence – position 5 does not come before position 6 in any sense the moves respect – so the recurrent variants are here to be measured, not because the shape of the data asks for them; measured, they diverge. "transformer" reads the pieces as a sequence and attends over them — a set, which is what attention takes, rather than the order the recurrent variants impose. It requires encoding = "piece".
rnn_units	Width of the recurrent layer ("lstm"/"gru")
n_blocks	Number of blocks: residual blocks under arch = "resnet", encoder blocks under "transformer". For the resnet every block is hidden[1] wide, since a skip connection has to add tensors of the same shape.
d_model	Model width (arch = "transformer" only). Must divide by n_heads.
n_heads	Attention heads (arch = "transformer" only)

ff_dim	Width of each block's feed-forward hidden layer (arch = "transformer" only). NULL, the default, means 4 * d_model, the usual ratio.
encoding	How a state reaches the network. "sticker" (default) reads it as 54 sticker positions through an embedding. "piece" reads it as DeepCubeA does — one one-hot per piece slot, saying which piece is in it and how it is turned — using cube_piece_layout, which also drops the pieces no move disturbs. It takes no embedding, being one-hot already, so embed_dim is ignored under it.
backend	"auto", "cpu" or "vulkan"

Value

List with value and policy models, plus the group

---

cube_adi_save *Save a Trained ADI Network*

Description

Writes the network to a directory: the two ggml models as their own files, and everything else as an RDS beside them.

Usage

```
cube_adi_save(net, path)
```

Arguments

net	A cube_adi_net
path	Directory to write into; created if absent

Details

The group cannot be written directly — it lives behind an external pointer that means nothing in a later session. What is saved instead is the material to rebuild it: the state length and, for each move, the permutation it performs, read back by applying the move to the identity. Loading replays those through perm_group and gets the same group, so the saved network does not depend on the function that happened to build it.

Value

path, invisibly

See Also

cube_adi_load

cube_adi_solve *Solve a Cube With a Trained Network*

Description

Walks downhill on the value network, one forward pass per move, refusing to revisit a state it has already stood on. Banning only the inverse of the last move is not enough – four of the same quarter turn also return to where they started – so the walk remembers where it has been, which covers every length of cycle with one rule.

Usage

```
cube_adi_solve(net, state, budget = 200L, use_policy = TRUE, batch_size = 256L)
```

Arguments

net	A trained cube_adi_net
state	Integer vector, the scrambled state
budget	Most moves to try before giving up
use_policy	Break ties with the policy network's ranking
batch_size	Batch the networks were trained with. Scoring reuses the compiled graph at this size, padding the 24 children out to fill it.

Value

List with solved, path (move names) and values

cube_adi_train *Train Value and Policy by Autodidactic Iteration*

Description

Train Value and Policy by Autodidactic Iteration

Usage

```
cube_adi_train(
  net,
  iterations = 100L,
  batch_states = 10000L,
  max_depth = 20L,
  epochs = 1L,
  batch_size = 256L,
  loss_thresh = 0.5,
  refresh_every = NULL,
  verbose = TRUE
)
```

Arguments

net	A cube_adi_net from cube_adi_model
iterations	Number of ADI iterations
batch_states	States generated per iteration
max_depth	Longest scramble drawn (uniform over 1..max_depth)
epochs	Passes over each batch
batch_size	Minibatch size for fitting and for scoring children
loss_thresh	Refresh the frozen network once the batch loss on value falls below this. Set it too low and the copy never refreshes: the targets stay pinned to an untrained network, every state looks one move from solved, and the mean target sticks near 1 while the value loss looks healthy. If the reported mean target stops rising, this is the first thing to raise. Ignored when refresh_every is given.
refresh_every	Refresh the frozen copy every this many iterations, instead of whenever the loss drops below loss_thresh. A threshold is awkward to set because it is compared against a loss whose scale changes as training proceeds: too low and the copy never refreshes, too high and it refreshes every iteration, which is worse. A copy refreshed every iteration is not frozen at all — the targets are then computed from the same weights that are being fitted to them, the network converges on agreeing with itself, and the mean target stops moving while the loss looks fine. A fixed interval sidesteps the question: the copy is genuinely held for that many iterations whatever the loss happens to be doing.
verbose	TRUE to report each iteration

Value

The network, with $history recording loss per iteration

---

cube_alg_table *The Algorithm Tables*

Description

The learned sequences the solvers look up: 57 OLL cases, 21 PLL cases, and the handful of small algorithms layer-by-layer repeats. Returned as written in the literature, with the expanded quarter-turn word alongside.

Usage

```
cube_alg_table(
  which = c("oll", "pll", "lbl_cross", "lbl_edge_perm", "lbl_corner_perm",
    "lbl_corner_twist")
)
```

Arguments

which	Character, one of "oll", "pll", "lbl_cross", "lbl_edge_perm", "lbl_corner_perm", "lbl_corner_twist"

Value

data.frame with name, notation, n_moves. The expanded words are on the "moves" attribute, one character vector per row.

See Also

cube_solve_cfop, cube_apply_word

Examples

```
oll <- cube_alg_table("oll")
nrow(oll)
head(oll)
```

---

cube_apply_word *Apply a Word of Moves to a Cube State*

Description

Applies moves written in standard notation. Half turns and wide turns are expanded into the package's quarter-turn alphabet, so "R U2 R'" is four moves, not three.

Usage

```
cube_apply_word(state, word)
```

Arguments

state	Integer vector of 54 stickers
word	Character, moves in standard notation ("R U R' U'")

Value

Integer vector of 54 stickers, the state after the word

See Also

cube_word_order, cube_read_state

Examples

```
s <- cube_identity(3)
# a quarter turn of U, four times, is the identity
identical(cube_apply_word(s, "U U U U"), s)
```

`cube_build_lslice` *Build the Layer Beside a Face, at Any Size*

Description

The second stage of Pochmann's centres, generalised: fill the layer that `cube_lslice_cells` names — the L face together with the column of each side face lying against it — so that every centre in it shows the colour of the face it sits on.

Usage

```
cube_build_lslice(state, n = NULL, max_rounds = 40L)
```

Arguments

`state`	Integer vector of $6n^2$ stickers.
`n`	Side of the cube. Inferred from the length of `state` when absent.
`max_rounds`	Most three-cycles to apply before giving up.

Value

List with components:

`state`	The cube after the moves
`path`	Character vector of moves
`count`	How many of the layer's centres are home
`target`	How many there are
`built`	Whether the layer was finished
`rounds`	How many three-cycles were used

Why a layer and not a face

Building face by face does not work, and the reason is measurable: every commutator that fills one face disturbs the four beside it, so a face finished early is taken apart later. A layer can be turned to the bottom and kept there, which is what makes the stage after this one safe.

Two stages, and both are needed

Bringing a piece into the layer and rearranging pieces already inside it are different problems, and one vocabulary does not answer both.

The ladder does the first: single moves of the whole alphabet, then pairs, then triples, taking the first word that raises the count. This is `l_slice_insert` of `src/cube_centres.h`, whose own note is that pairs carry the method — its worked example, an inner slice followed by a reversed U, runs 8, 8, 9, improving nothing on the first move, which is why words are judged whole rather than one move at a time.

Addressed three-cycles do the second. The ladder stops when the remaining pieces are inside the layer but in each other's places: no short word rearranges the layer's own contents, so the count simply stops improving, whatever the depth. A three-cycle moves exactly three centres and leaves the rest standing, and pointing one at named positions closes what the ladder cannot.

Measured over ten scrambles at each size, the ladder alone finished 2 of 30 and the two together 30 of 30 — reaching 17.9 of 20 on a 5x5x5 by itself and 20 of 20 with the cycles.

What the cycles need

A commutator for every moving orbit, and the 5x5x5 is the case that shows why: its plus centres are cycled only through the central slice. While that was excluded from `cube_centre_cycles` on the grounds that a central turn rotates the cube, half the pieces had no tool and builds stalled on them.

The triple of faces also has to be one a rotation can deliver. The commutators act on faces 0, 2 and 5, two of which are opposite, and rotation preserves adjacency — so twelve of the twenty triples are reachable and eight are not. In practice only reachable ones are ever wanted, which `cube_centres.h` reports measuring too.

See Also

`cube_lslice_cells`, `cube_centre_cycles`, `cube_centre_shots`

Examples

```
set.seed(1)
g <- cube_group(4)
s <- group_apply(g, group_identity(g),
                 sample(cube_move_names(4), 6, replace = TRUE))

r <- cube_build_lslice(s)
r$built
length(r$path)
```

`cube_central_moves` *Slice Turns That Reorient the Whole Cube*

Description

On an odd cube the middle layer of an axis carries that axis's fixed centres, so turning it is a rotation of the cube rather than a move within it. A method that means to keep a face where it is must leave these alone.

Usage

```
cube_central_moves(n)
```

Arguments

`n`	Side of the cube, 2 or more.

Details

Which moves these are is found by asking, not by counting layers: a move is central exactly when it disturbs the six fixed centres. An even cube has no fixed centres and so no central moves, and the answer is the empty vector.

Value

Character vector of move names, empty on an even cube.

See Also

cube_centre_shots, cube_centre_structure

Examples

```
cube_central_moves(5)   # 2x, 2y, 2z and their inverses
cube_central_moves(4)   # none: an even cube has no middle layer
```

---

cube_centre_counts *Count the Centre Pieces That Are Home*

Description

How many centre pieces show the colour of the face they sit on, counted per face. A face is finished when its count is the number of centres a face has at that size — four on a 4x4x4, nine on a 5x5x5, one on a 3x3x3.

Usage

```
cube_centre_counts(state, n = NULL, by_orbit = FALSE)
```

Arguments

state	Integer vector of $6n^2$ stickers.
n	Side of the cube. Inferred from the length of state when absent.
by_orbit	Whether to count each centre orbit separately. A 5x5x5 has three — the corner centres, the plus centres and the fixed six — and a method that solves them in turn needs to see them apart.

Details

The count compares a piece's colour against the face it began on, so it is only meaningful while the cube has not been turned as a whole. That is the state generate_state hands over and the one cube_solve_centres is given.

Value

With `by_orbit = FALSE`, an integer vector of six counts in face order U R F D L B. With `by_orbit = TRUE`, a `data.frame` of `orbit`, `face`, `home` and `of` — how many are home out of how many.

See Also

`cube_solve_centres`, `cube_centre_structure`

Examples

```
cube_centre_counts(cube_identity(4))
cube_centre_counts(cube_identity(5))

# the three orbits of a 5x5x5, seen apart
cube_centre_counts(cube_identity(5), by_orbit = TRUE)
```

---

cube_centre_cycles *Three-Cycles of Centres, One Per Orbit*

Description

The commutator that moves three centres and nothing else. Steps that fill a face leave the centres in arrangements a swap cannot reach — a three-cycle is an even permutation and a swap is odd — so a method needs this shape as well as the shots.

Usage

```
cube_centre_cycles(n, all = FALSE)
```

Arguments

`n`	Side of the cube, 4 or more. Smaller cubes have no movable centres and get no rows.
`all`	Whether to return every commutator found. The default keeps one per (orbit, face triple), which is what a method needs; `TRUE` keeps all of them.

Value

A `data.frame` with columns `word` (the eight moves, space separated), `orbit` (which centre orbit it cycles), `faces` (the three faces involved, comma separated, ascending) and `slots` (the slot on each, in that same face order).

Derived, and wider than the original

`src/cube_centres.h` carries one such commutator, written out for a 4x4x4: `1x U 2x' U' 1x' U 2x U'`, cycling a centre between faces 0, 2 and 5. The same shape is searched here instead — over every ordered pair of inner slices, about all three axes, conjugated by each of the six face turns. Each candidate is checked to move exactly three centres of one orbit before it is kept, so what comes back is a three-cycle by measurement rather than by construction.

The result is wider than the file's in two ways. Every moving orbit gets its own commutators, not just the one a 4x4x4 has; and they reach all twenty triples of faces rather than the single triple 0, 2, 5 that one fixed word acts on. Both matter to a method, which has to cycle whichever three faces the cube presents.

The central slice belongs in the search

An earlier version excluded `cube_central_moves` on the grounds that turning the middle layer of an odd cube turns the whole cube. That is true of the move by itself and false of a commutator built from it: only the whole word is judged, and the frame it disturbs in the middle it puts back at the end. Measured on a 5x5x5, 1644 of 2592 candidate words leave the fixed centres exactly where they were.

Excluding it cost the 5x5x5 an entire orbit. Its plus centres are cycled by `1x U 2x' U' 1x' U 2x U'` and nothing else of this shape — one character from the corner centres' own word, and that character is the central slice.

See Also

`cube_centre_shots`, `cube_centre_structure`, `cube_central_moves`

Examples

```
# a 4x4x4 has one moving orbit, cycled between many triples of faces
cyc4 <- cube_centre_cycles(4)
unique(cyc4$orbit)
length(unique(cyc4$faces))

# a 5x5x5 has two, and the plus centres need the central slice
cyc5 <- cube_centre_cycles(5)
unique(cyc5$orbit)
```

---

cube_centre_map_family

The Centres of a Cube of Any Size, Derived

Description

What a method needs to know about centres before it can build them: which stickers are centre stickers, which orbit each belongs to, where it sits on its face, and where every slice turn sends it. All of it read off `cube_moves` and `cube_orbits` rather than written down per size.

Why this is derived

The 4x4x4 centre solver in `src/cube_centres.h` carries its slice table as six hand-measured blocks of eight steps. That table is correct, and it is also the reason the file solves one size: the numbers say `f*16 + local`, the slots run 1 to 4, and both facts are assumptions about a 4x4x4 rather than about cubes.

Nothing in that table is independent of the permutations, though. A slice turn is a permutation; asking it where each centre sticker goes reproduces the hand table exactly — all six blocks, at `n = 4` — and asks the same question of any other size. That equality is checked in the tests, so the generic version is held to the hand-measured one rather than merely believed.

Slots

A face carries several centre stickers of the same orbit, and a method has to name them. They are numbered along the cycle of that face's own turn: slot 1 is the lowest sticker index in the group, slot 2 is where a quarter turn sends it, and so on. Numbering by the turn rather than by reading order means slot arithmetic is the same on every face and at every size — a quarter turn always adds one, modulo the cycle length.

See Also

`cube_centre_structure`, `cube_slice_map`, `cube_orbits`

`cube_centre_positions` *Sticker Positions of the Six Centres*

Description

The centres never move, which is what makes a face's name mean anything: the U face is the face whose centre is the U centre, whatever has happened to the rest of the cube.

Usage

```
cube_centre_positions()
```

Value

Integer vector of 6 sticker positions (1-based)

See Also

`cube_identity`

Examples

```
cube_centre_positions()
s <- cube_apply_word(cube_identity(3), "R U R' U'")
# centres are where they always were
all(s[cube_centre_positions()] == cube_centre_positions())
```

cube_centre_shots *Commutators That Carry Centres Off One Face and Spare Another*

Description

The generic form of the eight shots that `src/cube_centres5.h` lists by hand for a 5x5x5. A shot is a conjugation — a slice turn, a turn of the working face, then the slice back — and what makes it usable is what it does not do: the kept face's centres come back exactly where they were, so whatever has already been built there survives however many shots are fired afterwards.

Usage

```
cube_centre_shots(n, keep_face = NULL, turn_face = "U")
```

Arguments

`n`	Side of the cube, 2 or more.
`keep_face`	Face whose centres must not move, 0 to 5. Defaults to the face at the negative end of the y axis, which is D.
`turn_face`	Name of the face turn conjugated, default `"U"`.

Value

A `data.frame` with columns `word` (the three moves, space separated), `to_face` (the face centres are carried to), `n_moved` (how many centre stickers move) and `orbits` (which centre orbits they belong to, comma separated). No rows if the size has no moving centres.

What is required of a shot

Three conditions, each checked against the permutations rather than assumed. The kept face must be untouched. Some centre must actually cross between faces, which is what rules out the conjugations by a turn of the working face's own axis — they stir that face without emptying it. And the slice must not be a central one, since on an odd cube that turns the whole cube and moves the kept face out from under the method (`cube_central_moves`).

See Also

`cube_central_moves`, `cube_centre_structure`

Examples

```
# a 5x5x5 has eight, two onto each side face
s <- cube_centre_shots(5)
table(s$to_face)

# each carries both moving orbits at once, which is why the two are solved
# together rather than in turn
unique(s$orbits)
```

cube_centre_structure *Centre Stickers of an N x N x N Cube*

Description

One row per centre sticker: which face it is on, where on that face, which orbit it belongs to and which slot of its (face, orbit) group it occupies.

Usage

```
cube_centre_structure(n)
```

Arguments

n	Side of the cube, 2 or more.

Details

A centre piece carries exactly one sticker, so pieces and stickers are the same thing here and the table can be indexed either way.

Value

A `data.frame` with columns `sticker` (1-based index into the state vector), `face` (0 to 5), `local` (0-based position within the face), `orbit` (as numbered by `cube_orbits`) and `slot` (1-based, along the face-turn cycle).

See Also

`cube_slice_map`, `cube_centre_map_family`

Examples

```
# a 4x4x4 has one centre orbit, four per face
cs <- cube_centre_structure(4)
table(cs$orbit)

# a 5x5x5 has three: two of 24 and the six fixed centres
table(cube_centre_structure(5)$orbit)

# the fixed centres of an odd cube are alone in their group
cs5 <- cube_centre_structure(5)
table(cs5$slot[cs5$orbit == max(cs5$orbit)])
```

cube_colours *Colours and Positions: the Two Ways to Write a Cube Down*

Description

A cube state can record, for each sticker, either the colour it shows or the place it belongs in. The package works in positions — a state is a permutation of `1:(6n^2)`, and the solvers and searches all read it that way — while most outside sources, Kaggle's Santa 2023 among them, record colours.

Usage

```
cube_colours(state, n = NULL)

cube_colour_state(state, n = NULL)
```

Arguments

`state`	Integer vector of $6n^2$ entries: a permutation of `1:(6n^2)` for `cube_colours`, or colours for `cube_colour_state`. Colours may be `0..5` or `1..6`; the result uses whichever the input did.
`n`	Integer, the side of the cube; inferred from the length when omitted

Details

`cube_colours` goes from positions to colours, which always works and loses information. `cube_colour_state` goes back, which works only when the colours happen to determine the state.

Value

`cube_colours` returns $6n^2$ colours numbered from `0`, one per sticker, in face order U R F D L B. `cube_colour_state` returns a permutation of `1:(6n^2)`.

When colours are enough

On a 2x2x2 or a 3x3x3 every piece carries a different set of colours, so a colouring names exactly one state and the conversion is exact.

From 4x4x4 up it is not. The four wings of an edge share their two colours, and the centres of a face share one, so a colouring cannot say which of them is which — and the number of states it stands for grows quickly with n. `cube_colour_state` still returns a state, choosing among the identical pieces in the order they are numbered, but it warns; the state it returns shows the right colours and is not otherwise the state you started from.

See Also

`cube_santa_state` for the same journey across notations, `cube_is_colour_solved`, `cube_pieces`

Examples

```
set.seed(1)
s <- generate_state(group = cube_group(3), n_moves = 10)

# positions to colours and back is exact on a 3x3x3
identical(cube_colour_state(cube_colours(s)), s)

# a solved cube is six blocks of one colour
table(cube_colours(cube_identity(3)))
```

`cube_edge_algs` *The Published Edge-Pairing Algorithms, at Any Size*

Description

The six slice-flip-slice words the 4x4x4 solver uses, written in the package alphabet for a cube of side `n`. A slice brings two pieces of one edge together, a face turn flips one of them, the slice goes back.

Usage

```
cube_edge_algs(n)
```

Arguments

`n`	Side of the cube, 3 or more.

Value

A character vector of six words, each a space-separated sequence of package moves.

Why these six

They are the set measured in `src/cube_edges.h`, which records for each one its length, how many pairs it breaks and how many slots it touches. All six leave the centres built, which is what makes them safe to run after the centre stage. The last is the parity case — it swaps two pieces of one edge rather than permuting whole edges.

What changes with size

Only the wide turns, and only in which numbered slice they name. The words are stated here in the sources' own notation (`Rw`, `Dw`, `Uw`) and expanded through `cube_wide_turn`, so a single statement covers every size instead of one transcription per cube. At `n = 4` the expansion reproduces the six constants in the reference file exactly, which the tests check.

Verified at `n = 4` against the reference's own table: from a solved cube the six leave 9, 9, 10, 10, 10 and 8 edges whole — that is 3, 3, 2, 2, 2 and 4 pairs broken, as the reference records.

See Also

cube_wide_turn, cube_edge_counts

Examples

```
cube_edge_algs(4)[1]
cube_edge_algs(5)[1]     # the same algorithm, the slice renumbered
```

---

cube_edge_counts *How Many Edges Are Whole*

Description

The count a pairing stage ranks its moves by, and the per-edge breakdown behind it. An edge is whole when every piece sitting on it shows the same pair of colours — on a 4x4x4 that is two wings agreeing, on a 5x5x5 three pieces, on an odd cube the middle piece counts with its wings.

Usage

```
cube_edge_counts(state, n = NULL, structure = NULL)
```

Arguments

state	Integer vector of $6n^2$ stickers.
n	Side of the cube; inferred from state when NULL.
structure	Optional result of cube_edge_structure for the same n, to avoid recomputing it in a loop.

Details

This is the rule cube_is_reduced applies to decide the whole cube; here it is asked one edge at a time, because a solver that can only see "reduced or not" has nothing to rank a candidate move by.

Value

A list with whole (how many of the twelve edges are whole), total (twelve) and by_edge, a logical vector one entry per edge.

See Also

cube_edge_structure, cube_is_reduced

Examples

```
cube_edge_counts(cube_identity(5))$whole      # 12

# an inner slice splits the edges it touches
s <- cube_identity(5)
cube_edge_counts(s[cube_moves(5)[["1x"]]])$whole
```

cube_edge_map *Where a Turn Sends Each Edge Piece*

Description

The permutation a single move induces on the edge slots: which slot each piece's contents lands in. This is the table an edge solver reads instead of searching, and it is the generic form of `slot_maps()` in `src/cube_edges.h`.

Usage

```
cube_edge_map(n, move, structure = NULL)
```

Arguments

`n`	Side of the cube, 3 or more.
`move`	Name of a move, as given by `cube_moves`.
`structure`	Optional result of `cube_edge_structure` for the same `n`, to avoid recomputing it in a loop.

Details

Only the slots the move actually disturbs are listed. Measured at `n = 4` and `n = 5`, an outer turn moves `4(n - 2)` slots and never splits an edge, while a slice moves 4 and splits all four it touches — see `cube_edge_map_family` for why that difference is the whole mechanism of pairing.

Value

A `data.frame` with one row per disturbed slot and columns `from_piece`, `from_edge`, `from_pos`, `to_piece`, `to_edge`, `to_pos` and `splits` — the last `TRUE` when the move carries this piece off an edge whose other pieces it leaves behind.

See Also

`cube_edge_structure`, `cube_slice_map`

Examples

```
cube_edge_map(4, "U")     # eight slots, four whole edges
cube_edge_map(4, "1x")    # four slots, four edges split
```

`cube_edge_map_family` *The Edges of a Cube of Any Size, Derived*

Description

What a method needs to know about edges before it can pair them: which pieces are edge pieces, which of the twelve cube edges each sits on, where along that edge it lies, and where every turn sends it. All of it read off `cube_pieces` and `cube_moves` rather than written down per size.

Why this is derived

The 4x4x4 edge solver in `src/cube_edges.h` carries its slots as a hand-measured table of twenty-four sticker pairs, ordered "the twelve edges by face pair, A then B". That table is correct, and it is also the reason the file solves one size: twenty-four is a fact about a 4x4x4, and so is the pair of names A and B. A 5x5x5 puts three pieces on every edge, a 7x7x7 puts five, and neither has halves to call A and B.

Nothing in the table is independent of the geometry, though. An edge piece is a piece with two stickers; the edge it belongs to is the pair of faces it touches; its place along that edge is the one coordinate that is not pinned to a face. Asking those three questions of `cube_pieces` reproduces the hand table at `n = 4` and asks the same question of any other size. That equality is checked in the tests, so the generic version is held to the hand-measured one rather than merely believed.

How many pieces an edge holds

Measured, not assumed: `n = 3` through `n = 7` give 12 cube edges at every size, holding `n - 2` pieces each — 1, 2, 3, 4, 5. The pieces fall into `floor((n - 1) / 2)` orbits, and on an odd cube the last of those holds the 12 middle edges, one per cube edge, sitting on the symmetry plane. Those are the pieces that behave like the edges of a 3x3x3.

Why outer turns cannot pair anything

The mechanism the 4x4x4 solver is built on generalises exactly. Measured at `n = 4` and `n = 5` over the whole alphabet:

- the six OUTER turns move 4(n-2) slots and touch 4 cube edges, none of them partially — an edge travels whole, so an outer turn can neither pair anything nor break a pair;
- every SLICE moves 4 slots across 4 cube edges, all four partially — it carries one piece off each edge and leaves the rest. That mismatch is the entire mechanism of pairing, and it is also why a slice on its own wrecks the centres.

So a slice has to be wrapped: open with it, work with outer turns, close with its inverse. That is the same shape the l-slice stage of the centres used.

See Also

`cube_edge_structure`, `cube_edge_map`, `cube_pieces`, `cube_centre_structure`

cube_edge_structure *Edge Pieces of an N x N x N Cube*

Description

One row per edge piece: which of the twelve cube edges it sits on, where along that edge it lies, which orbit it belongs to and which stickers it carries.

Usage

```
cube_edge_structure(n)
```

Arguments

n	Side of the cube, 3 or more.

Details

An edge piece carries exactly two stickers, which is what distinguishes it from a corner (three) and a centre (one). See cube_edge_map_family for why this is derived rather than tabulated per size.

Value

A data.frame with one row per edge piece and columns piece, edge, face_lo, face_hi, pos, slot, orbit, label, sticker_a and sticker_b — the last two 1-based indices into the state vector.

Naming an edge

A cube edge is named by the two faces it lies between, as face_lo and face_hi with face_lo < face_hi, so the name does not depend on which way the edge is read. The edge column numbers those twelve pairs in ascending order, and is stable at every size.

Naming a place along an edge

pos is the coordinate that runs along the edge, from 1 to n - 2. slot is the same thing numbered from the nearer end, from 1 up, so that pieces which are mirror images of each other across the middle of the edge share a slot number; on an odd cube the middle piece is the only one of its slot. Pieces of one edge with the same slot are the ones a solver has to tell apart, and pos is what tells them.

See Also

cube_edge_map, cube_pieces

Examples

```
e <- cube_edge_structure(4)
nrow(e)                  # 24 wings, two to an edge
table(e$edge)            # twelve edges, two pieces each

e5 <- cube_edge_structure(5)
nrow(e5)                 # 36: 24 wings and 12 middles
table(e5$orbit)
```

`cube_empty_u_slice` *Empty the Top Face onto the Sides*

Description

The third stage of Pochmann's centres, generalised. With a layer built by `cube_build_lslice` and turned to the bottom, the shots of `cube_centre_shots` fire pieces off U onto the side faces they belong to. Every shot leaves D untouched, so what has been built and turned down survives however many shots follow.

Usage

```
cube_empty_u_slice(state, n = NULL, max_shots = 200L)
```

Arguments

`state`	Integer vector of $6n^2$ stickers, with a layer already built and turned to the bottom.
`n`	Side of the cube. Inferred from the length of `state` when absent.
`max_shots`	Most shots to fire before giving up.

Value

List with components:

`state`	The cube after the moves
`path`	Character vector of moves
`count`	How many movable centres are home
`target`	How many there are
`faces`	How many faces are finished
`shots`	How many shots were fired
`reloads`	How many of those were reloads

How a shot is chosen

The piece is named first and the shot chosen for it, not the other way round: a centre on U belonging to a side face, a shot firing at that face, and the number of U turns that bring the piece to that shot's entry slot.

Asking instead for a shot that raises the count refuses the ones that work. Measured on the positions where an earlier version stalled: every foreign piece had two shot-and-setup pairs that sent it home, and not one of the thirty-two combinations raised the total. A piece going home displaces one that was there already, and the count recovers a shot or two later.

A shot has a second use as well. It is a 4-cycle — it takes a piece off U and pulls another up in exchange — so once U holds only pieces belonging to U, the way on is to spend one to bring up another. Such a reload lowers the count by design and is judged by the ammunition instead: how many pieces on U belong to neither the top face nor the bottom.

What it reaches

Measured over five scrambles of a 5x5x5, it takes the cube from 25 centres home to about 36 of 48, closing one face. It does not finish: the last two or three pieces on U can be placed only at the cost of a side face that is unfinished too. Pochmann answers that with a tilt — turning an unfinished face up to U — but a tilt is only legal once three faces are done, and from one it cannot be reached. Widening the guard was tried and lost ground (30.4 centres against 35.4), and so was ranking by a target face (36.2).

See Also

cube_build_lslice, cube_centre_shots, cube_centre_counts

Examples

```
set.seed(1)
g <- cube_group(5)
s <- group_apply(g, group_identity(g),
                 sample(cube_move_names(5), 4, replace = TRUE))

lay <- cube_build_lslice(s)
down <- group_apply(g, lay$state, cube_expand_word("z'", 5))
r <- cube_empty_u_slice(down)
r$count
r$faces
```

---

cube_expand_alg *Expand an Algorithm into the Package's Alphabet*

Description

Published algorithms are written with half turns (R2), wide turns (r), and whole-cube rotations (x, y, z). The package's alphabet has none of those: it is quarter turns of single layers. This is the translation.

Usage

```
cube_expand_alg(notation)
```

Arguments

notation	Character, an algorithm in standard notation

Details

Rotations are the interesting case. A rotation in a printed algorithm is an instruction to the reader — turn the cube, and read what follows from where it now is — so it is expanded by renaming the letters after it, not by turning the cube and leaving them alone. Expanding it the other way silently produces a different algorithm, which is what happened to the published V-perm before this was written.

Value

Character vector of move names in the package alphabet

See Also

cube_alg_table, cube_apply_word

Examples

```
cube_expand_alg("R U2 R'")        # the half turn becomes two moves
cube_expand_alg("y R U R'")       # the rotation renames what follows
```

---

cube_expand_move *Wide Turns and Whole-Cube Rotations*

Description

The alphabet a cube is generated in turns one layer at a time, because that is what keeps the metric honest: $6n$ moves, half turns written out, so a shortest path counts quarter turns. Speedcubing writes in a larger vocabulary. `Rw` turns the two layers nearest the right face together, `3Rw` turns three, and `x` turns the whole cube. Reduction methods for the 4x4x4 and up are published in that vocabulary and in no other.

Usage

```
cube_expand_move(name, n)

cube_expand_word(word, n)

cube_wide_move(name, n)

cube_wide_word(word, n)
```

Arguments

name	Single move name, such as "Rw", "3Rw'", "x2" or "R"
n	Integer, the side of the cube (at least 2)
word	A word: a character vector of names, or one space-separated string

Details

These functions translate it. cube_expand_move takes one such name and returns the layer moves it stands for; cube_expand_word does a whole algorithm; cube_wide_move returns the single permutation the name performs. Nothing here enters the generating set — Rw is a word of two moves, not a move — so the metric and every diameter measured in it are untouched.

Value

cube_expand_move and cube_expand_word return a character vector of layer moves from the cube's own alphabet; cube_wide_move returns a permutation of 1:(6n^2).

The notation

R	the face layer alone
Rw, r	the face layer and the one behind it
3Rw	three layers, counting in from the face
3R	the third layer alone, counting in from the face
x	every layer: the whole cube turned
M, E, S	the 3x3x3 slices, where $n = 3$

A trailing prime inverts and a trailing 2 doubles, as usual, and a half turn expands into two quarter turns because the alphabet has no other way to say it.

A block move counts slices, and the count is bounded at both ends by the WCA regulations (12a2, 12a2+): strictly more than one and strictly fewer than n. One slice is a face turn and drops the w; every slice is a rotation and is written x, y or z. So "1Rw" names nothing on any cube, "4Rw" names nothing on a 4x4x4, and both are refused — otherwise one move would have two spellings. Rw and 2Rw are the same move, as the regulations say.

The senses are the ones the literature uses: x follows R, y follows U, z follows F. Lower-case r is accepted as a synonym for Rw, which is how many sources write it.

Why not generators

A wide turn could be added to the alphabet instead, and then Rw would cost one move rather than two. That would change what a shortest path is, and the numbers this package reports — diameters, level sizes, the distance any search returns — would no longer be comparable with the ones it reported before, nor with the published quarter-turn figures. Keeping the vocabulary outside the alphabet costs nothing and avoids all of that.

See Also

cube_moves, cube_move_names, cube_group

Examples

```
# on a 4x4x4 a wide turn is two layers
cube_expand_move("Rw", 4)

# three of them, and the whole cube
cube_expand_move("3Rw", 4)
cube_expand_move("x", 4)

# a single inner layer, counted in from the face
cube_expand_move("3R", 5)

# a whole algorithm at once
cube_expand_word("Rw U Rw'", 4)

# and the permutation it performs
p <- cube_wide_move("Rw", 4)
length(p)
```

---

cube_group *The Group of an N x N x N Cube*

Description

Builds the cube as a perm_group, so every search function in the package runs over it: the same BFS and the same store that work on TopSpin work here, and nothing about them knows the size of the cube.

Usage

```
cube_group(n, moves = NULL)
```

Arguments

n	Integer, the side of the cube (at least 2)
moves	Character vector naming the alphabet, a subset of cube_move_names(n) (default all $6n$ of them). A subset is a legitimate subgroup to search — faces only, or one axis only.

Value

An object of class perm_group

See Also

cube_group for the 3x3x3 with half turns as single moves, cube_nnn, perm_group

Examples

```
g <- cube_group(3)
g
group_moves(g)

# face turns only: the slices left out
cube_group(3, moves = c("U", "U'", "R", "R'", "F", "F'",
                        "D", "D'", "L", "L'", "B", "B'"))

# a 2x2x2 is small enough to walk about in
g2 <- cube_group(2)
group_order(g2, "R")
group_order(g2, "R U")
```

`cube_identity` *Solved State of an N x N x N Cube*

Description

Solved State of an N x N x N Cube

Usage

```
cube_identity(n)
```

Arguments

`n`	Integer, the side of the cube

Value

Integer vector `1:(6*n^2)`

See Also

`cube_is_colour_solved`

Examples

```
length(cube_identity(3))
length(cube_identity(4))
```

`cube_is_colour_solved` *Test Whether Every Face of a Cube is a Single Colour*

Description

A sticker's colour is the face it started on, so this asks whether each face now carries stickers of one colour — not whether they are back in their original positions.

Usage

```
cube_is_colour_solved(state, n = NULL)
```

Arguments

`state`	Integer vector of length $6n^2$
`n`	Integer, the side of the cube. Inferred from `state` when absent.

Details

The distinction only appears once slice turns are in play. Face turns leave the centres alone, so from `cube_identity` they can never reach a state that is one-coloured without being the identity outright. Slices do move the centres, and then a cube turned bodily in space is solved by any reasonable reading while its sticker vector is not `1:6n^2`.

Value

Logical

See Also

`cube_identity`, `cube_nnn`

Examples

```
cube_is_colour_solved(cube_identity(3))

# a face turn breaks it
g <- cube_group(3)
cube_is_colour_solved(group_apply(g, cube_identity(3), "R"))

# turning every layer of one axis is turning the whole cube: the colours
# stay solved, but the stickers are no longer where they started
s <- group_apply(g, cube_identity(3), c("R", "M'", "L'"))
cube_is_colour_solved(s)
identical(s, cube_identity(3))
```

`cube_is_reduced` *Is a Cube Reduced?*

Description

Reduction is the state a large cube has to reach before a 3x3x3 method can touch it: every face's centres one colour, so a face acts as one, and the wings of every edge a matching group, so an edge acts as one. A cube in that state is a 3x3x3 with fat pieces.

Usage

```
cube_is_reduced(state, n = NULL)
```

Arguments

`state`	Integer vector of $6n^2$ stickers — from `generate_state` with `group = cube_group(n)`.
`n`	Side of the cube. Inferred from the length of `state` when absent.

Details

What this does *not* ask is where anything is. A solved cube turned by one `U` is still reduced — the centres are whole and the pairs are together, they are merely in the wrong place, and putting them right is the 3x3x3's job rather than the reduction's. Asking for position here is the easy mistake and it makes the test reject cubes that are plainly reduced.

Value

`TRUE` if the cube is reduced.

Any size

Nothing in the test is about a 4x4x4 beyond how many pieces there are. A face has $(n-2)^2$ centres rather than four and an edge has $n-2$ wings rather than two, and both numbers come from `cube_pieces`. An odd cube reduces too: its middle edges and fixed centres are groups of one, which pass the test by having nothing to disagree with.

A 3x3x3 is reduced in every state, and that is the right answer rather than a degenerate one — there is nothing to reduce.

See Also

`cube_solve4`, `cube_pieces`, `cube_centre_counts`

Examples

```
s <- cube_identity(4)
cube_is_reduced(s)

# a turned face is still reduced: the pieces are whole, just not home
m <- cube_moves(4)
names(m) <- cube_move_names(4)
cube_is_reduced(s[m[["U"]]])

# an inner-layer turn is not: it splits the pairs and the centres
cube_is_reduced(s[m[["1x"]]])

# a 5x5x5 works the same way
cube_is_reduced(cube_identity(5))
```

`cube_kociemba` *Solve a 3x3x3 by Two-Phase Search*

Description

The fifth solver in the package and the only one that is not a human method. The other four place pieces by rule or by table and stay inside what a person could do by looking; this one searches, and pays for the short solution in a way no cuber could.

Usage

```
cube_kociemba(state, max_depth1 = 12L, max_depth2 = 18L, node_budget = 5e+07)
```

Arguments

`state`	Integer vector of 54 stickers, a reachable cube state — from `generate_state` with `group = cube_group(3)`.
`max_depth1, max_depth2`	Deepest search each phase may go to. The defaults are past what either needs.
`node_budget`	Nodes a phase may visit before giving up. A search that stops here returns nothing rather than a wrong answer; see `cube_kociemba_report` for which phase it was.

Value

List with `path`, the character vector of moves, and `found`, whether a solution was reached within the limits. The same shape as `cube_solve_cfop` and the other 3x3x3 solvers, so the five can be used interchangeably.

How this differs from the published algorithm

Kociemba's own two-phase algorithm does not stop at the first solution. It keeps searching phase 1 for other ways into the subgroup and keeps whichever leaves phase 2 shortest, which is how it reaches around twenty moves; the two phases are cheap enough to run many times over. This takes the first entry phase 1 finds, so a cube one quarter turn from solved comes back as `R R R` rather than `R`-prime. The prune tables are hash tables grown on demand rather than full coordinate tables. Short rather than shortest, in other words, and the demo names it `KociembaMod` for that reason.

Where the search machinery comes from

The engine underneath – the canonical move automaton, the hash prune table grown to half the depth about to be searched, the two-level cutoff that abandons a whole move class when the bound is hopeless by more than one – follows `twips` (https://github.com/cubing/twips) by Lucas Garron and the cubing.js authors, MPL-2.0. The phase definitions and the piece reading are this package's own, built on its existing sticker geometry.

The algorithm is the observation that a cube is much easier to solve if you first make it easy. The subgroup $G_1 = \langle U, D, L^2, R^2, F^2, B^2 \rangle$ is the set of states reachable without ever quarter-turning L, R, F or B, and it is small: about 20 million states against 43 quintillion. Getting into it is the first phase; finishing inside it is the second.

What makes a state a member of G_1 is three things, and each is exactly what the excluded quarter turns would break: every edge oriented, every corner oriented, and the four E-slice edges somewhere in the E slice — though which of the four sits where does not matter yet. Phase 1 searches for that and nothing else, on a coordinate that has thrown the rest of the cube away.

Why the moves are counted twice over

The returned path is in the package's quarter-turn alphabet, so a half turn comes back as two moves. Inside the solver it is one: phase 2 is generated by $\langle U, D, L^2, R^2, F^2, B^2 \rangle$ and a half turn there is a single step of the group, not a shorthand for two. Move cost is a property of the phase rather than of the package, which is what lets one search engine serve both phases.

Centres

Slice moves turn the centres, and a piece is identified by its colours read against them — so on a cube whose centres have moved, every cubie reads as the wrong one. As in the four human methods, the cube is turned bodily back first and the rotation is returned as part of the path, so the answer applies to the cube as it was handed over.

See Also

`cube_solve_cfop` and `cube_solve_lbl` for the methods that look at the cube, `cube_kociemba_report` for what the last solve did

Examples

```
set.seed(42)
s <- generate_state(group = cube_group(3), n_moves = 10)

res <- cube_kociemba(s)
```

```
res$found
length(res$path)

# it really solves it
moves <- cube_moves(3)
names(moves) <- cube_move_names(3)
for (mv in res$path) s <- s[moves[[mv]]]
identical(s, cube_identity(3))
```

`cube_kociemba_init` *Build the Kociemba Prune Tables*

Description

The tables grow themselves as the search asks for depth, so nothing needs calling before `cube_kociemba`. This exists for the caller who would rather pay that cost somewhere they expect it than inside the first solve.

Usage

```
cube_kociemba_init(
  table1 = 4194304,
  depth1 = 0L,
  table2 = 16777216,
  depth2 = 0L
)
```

Arguments

`table1`, `table2`	Size in entries of each phase's table, rounded up to a power of two.
`depth1`, `depth2`	Depth to fill in now. Zero seeds the table with the goal alone and leaves the rest to the search.

Value

Invisibly `NULL`, called for its effect.

See Also

`cube_kociemba`

Examples

```
cube_kociemba_init()
```

cube_kociemba_report *What the Last Kociemba Solve Did*

Description

"No solution" and "ran out of budget" are different facts, and a caller that cannot tell them apart goes looking for a fault in the cube when the answer is to raise a limit. This reports the outcome of each phase of the most recent cube_kociemba call, and how many nodes it visited.

Usage

```
cube_kociemba_report()
```

Value

List with phase1, phase2 — each one of "found", "no_solution" or "exhausted" — and phase1_nodes, phase2_nodes.

See Also

cube_kociemba

Examples

```
set.seed(42)
s <- generate_state(group = cube_group(3), n_moves = 10)
cube_kociemba(s)
cube_kociemba_report()
```

cube_kociemba4 *Solve a 4x4x4 by Four-Phase Search*

Description

The three reduction phases of cube_kociemba4_reduce, and then the cube — now a 3x3x3 with fat pieces — handed to cube_kociemba. That is the fourth phase, and it is the only one that lives on this side of the C++ boundary: the reduced cube is squeezed from 96 stickers to 54, solved, and the answer lifted back to 4x4x4 moves.

Usage

```
cube_kociemba4(
  state,
  max_depth1 = 10L,
  max_depth2 = 12L,
  max_depth3 = 14L,
  node_budget = 5e+07,
  progress_every = 0,
  stop_at_first = TRUE,
  budget_steps = c(0.1, 1),
  max_orientations = length(.cube4_orientations)
)
```

Arguments

`state`	Integer vector of 96 stickers, a 4x4x4 state.
`max_depth1`, `max_depth2`, `max_depth3`	Deepest each reduction phase may search to.
`node_budget`	Nodes each phase may visit before giving up.
`progress_every`	Print a line every this many nodes, naming the phase, the depth being searched and the count so far; 0 for silence. Phase 3 can spend tens of millions of nodes on a hard cube, which is minutes of no output at all, and a caller cannot tell that from a hang without this.
`stop_at_first`, `budget_steps`, `max_orientations`	Passed to `cube_kociemba4_reduce`: whether to accept the first rotation that reduces the cube rather than the shortest of all of them, the rungs of the node budget the rotations are swept at, and how many rotations to try. The defaults favour time over length, because the exhaustive form pays the full node budget once per rotation.

Details

The same limits apply as to the reduction — see *What this reaches today* in `cube_kociemba4_reduce`. Where this returns a path it is a short one, a few dozen moves against the several hundred `cube_solve4` takes; where it returns nothing, `cube_solve4` is what to call.

Value

List with components:

`path`	Character vector of moves, the whole solution
`found`	Logical, whether the cube ended solved
`reduction`	Character vector, the part of the path that reduced it
`cube3`	Character vector, the part that solved the reduced cube
`failure`	Empty string, or which step did not finish

See Also

cube_kociemba4_reduce, cube_kociemba4_report, cube_solve4

---

cube_kociemba4_reduce *Reduce a 4x4x4 by Four-Phase Search*

Description

Reduction is what an even cube has to reach before a 3x3x3 method can touch it: the four centres of a face one colour, so the face acts as one, and the two wings of every edge together, so the edge acts as one. This searches for it, in the three phases described in `src/kociemba4.h`: the F/B centres, then the other four faces' centres, then the wings.

Usage

```
cube_kociemba4_reduce(
  state,
  max_depth1 = 10L,
  max_depth2 = 12L,
  max_depth3 = 14L,
  node_budget = 5e+07,
  progress_every = 0,
  prune_depth_bonus = 0L,
  orientations = .cube4_orientations,
  stop_at_first = TRUE,
  budget_steps = c(0.1, 1),
  max_orientations = length(orientations)
)
```

Arguments

`state`	Integer vector of 96 stickers, a 4x4x4 state — from `generate_state` with `group = cube_group(4)`.
`max_depth1`, `max_depth2`, `max_depth3`	Deepest each reduction phase may search to.
`node_budget`	Nodes a phase may visit before giving up. A phase that stops here makes the whole call return an empty path rather than a wrong one; `cube_kociemba4_report` says which phase it was.
`progress_every`	Print a line every this many nodes, naming the phase, the depth being searched and the nodes so far. Zero, the default, prints nothing. A hard scramble can spend a minute inside phase 3, where silence is indistinguishable from a hang; this is for telling them apart.
`prune_depth_bonus`	Levels to build phase 3's prune table beyond the usual half of the search depth. Zero is the standard rule. Building deeper costs time up front and sharpens the heuristic; this is the dial for measuring whether that trade pays on a given cube.

`orientations` Whole-cube rotations to try, as words. The reduction is run once per entry and the cube rotated back afterwards, because phases 1 and 2 cannot see the wings: among their equally short routes some leave phase 3 one move from done and others leave it twenty-eight out, and nothing in those phases prefers either. Defaults to all 24.

`stop_at_first` Return the first orientation that reduces the cube, rather than trying them all and keeping the shortest path. `TRUE` by default: every orientation costs a full reduction, and one that cannot finish spends the whole `node_budget` before saying so, so the exhaustive form can cost twenty-four times the budget on a single cube. Set `FALSE` for the shortest answer the rotations can give, and expect it to take considerably longer.

`budget_steps` Rungs of the node budget, as fractions of `node_budget`, used when `stop_at_first` is `TRUE`. Every orientation is tried at the first rung before any is tried at the second, which keeps a cheap success cheap whichever rotation it needed – an orientation only spends the full budget once all of them have failed at less. Ignored when `stop_at_first` is `FALSE`, where comparing path lengths requires every orientation to have run at the same budget. Set to `1` for a single rung at the full budget. See `R/budget_ladder.R` for the measurements.

`max_orientations` Try only this many of `orientations`. Fewer is faster in direct proportion; the risk is that a cube whose only workable rotation was cut off stops reducing at all.

Value

List with `path`, the moves taking the cube to a reduced state, and `found`, whether the search got there within the limits. The same shape the package's other solvers return, so a caller can read `$found` without knowing which one it called.

Parity

Reduction can leave a cube in a state no 3x3x3 reaches — the wings paired and the corners solved with the permutation odd — and the package's other 4x4x4 solver, `cube_solve4`, meets that at the end and repairs it with an inner-layer algorithm. Here it cannot arise: the parity is carried in phase 3's coordinate, so a solution that would leave the cube parity-odd is not a solution to the phase and the search never returns it. No repair step follows, because there is nothing to repair.

What this reaches today

Phase 3's coordinate grows by about a factor of fifteen per level and its search is the deep one. Measured over five scrambles at each length with the default budget: five moves reduces every time in about a tenth of a second, six moves three times in five, seven moves once in five. Past that the phase spends its budget and the function returns nothing — which `cube_kociemba4_report` distinguishes from "no solution exists". Raising `node_budget` buys some of this back at a cost in time. For a cube from a full scramble, `cube_solve4` is the one that finishes; it takes many more moves and always gets there.

See Also

cube_kociemba4 for the whole solve, cube_is_reduced for the test this aims at, cube_solve4 for the solver that always finishes

---

cube_kociemba4_report *What the Last 4x4x4 Reduction Did*

Description

As cube_kociemba_report is to the 3x3x3, so this is to the three reduction phases: which of them finished, which ran out, and what each cost. A phase that reports "exhausted" is one to raise node_budget for; "no_solution" within the depth given is a different fact and wants max_depth raised instead.

Usage

```
cube_kociemba4_report()
```

Details

Phase 4 does not appear here. It is cube_kociemba on the squeezed cube, and cube_kociemba_report is what reports it.

Value

List with phase1, phase2, phase3 — each one of "found", "no_solution" or "exhausted" — and phase1_nodes, phase2_nodes, phase3_nodes.

See Also

cube_kociemba4_reduce, cube_kociemba4

---

cube_layer_move *A Single Layer Turn, by Axis and Layer*

Description

The generator's own vocabulary, for when naming a layer is less convenient than pointing at it — on a 5x5x5, say, where the inner layers have no letters anyone agrees on.

Usage

```
cube_layer_move(n, axis, layer, turns = 1L)
```

Arguments

n	Integer, the side of the cube
axis	Integer, 1, 2 or 3 for the x, y and z axes. x runs L to R, y runs D to U, z runs B to F.
layer	Integer in 1:n, which layer along that axis turns. Layer 1 is at the negative end, layer n at the positive one.
turns	Integer, quarter turns clockwise seen from the positive end of the axis: 1, 2 or 3. Half turns are available here even though they are not in the alphabet.

Value

Integer vector of length $6n^2$, the permutation

See Also

cube_moves

Examples

```
# on a 3x3x3, layer 3 about y is the U face
identical(cube_layer_move(3, axis = 2, layer = 3, turns = 3),
          cube_moves(3)[["U"]])

# layer 2 about x is the M slice
identical(cube_layer_move(3, axis = 1, layer = 2, turns = 1),
          cube_moves(3)[["M"]])
```

---

cube_lslice_cells *The Layer Beside a Face, and What Leaves It Alone*

Description

Building centres face by face does not work: the commutators that fill one face disturb the four beside it, so a face finished early is taken apart later. Pochmann's method builds a *layer* instead — one face together with the column of each side face lying against it — because a layer can be turned to the bottom and kept there.

Usage

```
cube_lslice_cells(n)

cube_free_moves(n)
```

Arguments

n	Side of the cube, 2 or more.

Details

cube_lslice_cells is that layer, and cube_free_moves is the vocabulary that may be used once it is built: the single moves that leave every one of its centres where it stands.

Value

For cube_lslice_cells, a data.frame of the layer's centres with columns sticker, face, orbit and slot. For cube_free_moves, a character vector of move names.

Derived, not tabulated

src/cube_centres.h writes the layer out as twelve hand-measured cells and names the free moves in a comment. Both are read off the permutations here instead: the layer is what a wide L turn moves, and a move is free when it moves none of it. At n = 4 the two agree exactly — twelve cells, and the free moves R, R'}, \code{2x}, \code{2x' — which the tests check. At other sizes the same questions have bigger answers: twenty cells on a 5x5x5, thirty-two on a 6x6x6.

See Also

cube_centre_shots, cube_centre_structure

Examples

```
# the twelve centres of a 4x4x4's l-slice: all of L, two of each side face
table(cube_lslice_cells(4)$face)

# what may be turned once it is built
cube_free_moves(4)

# the same questions at another size
nrow(cube_lslice_cells(5))
cube_free_moves(5)
```

---

cube_move_names *Move Names of an N x N x N Cube*

Description

Move Names of an N x N x N Cube

Usage

```
cube_move_names(n)
```

Arguments

n	Integer, the side of the cube (at least 2)

Value

Character vector of $6n$ move names, in table order

See Also

cube_moves

Examples

```
cube_move_names(2)
cube_move_names(3)
cube_move_names(4)
```

---

cube_moves *Move Table of an N x N x N Cube*

Description

The $6n$ quarter turns of a cube of side n, as permutations of its $6n^2$ stickers. Generated from the geometry rather than tabulated, so the same rule gives the 2x2x2, the 3x3x3 and anything larger.

Usage

```
cube_moves(n)
```

Arguments

n	Integer, the side of the cube (at least 2)

Value

Named list of integer vectors, each a permutation of `1:(6*n^2)` read as `new[i] <- state[perm[i]]`

See Also

cube_group, cube_nnn

Examples

```
m <- cube_moves(3)
names(m)

# the twelve face turns agree with the hand-written 3x3x3 table
identical(m[["R"]], cube_moves(3)[["R"]])

# a 4x4x4 has two inner layers per axis
names(cube_moves(4))
```

cube_nnn *Cubes of Any Size*

Description

The 3x3x3 is one member of a family, and the family is easier to describe than any single member of it. A move is an axis, a layer along that axis and a quarter turn; that vocabulary covers every cube, and what changes from one size to the next is only how many layers there are.

Details

On a 3x3x3 the three layers of an axis are the two faces and the slice between them — `L`, `M`, `R` going along x — so the slice turns come out of the same rule as the face turns rather than being added separately. On a 4x4x4 there are two inner layers per axis, and on a cube of side n there are $n - 2$.

Stickers

A state is an integer vector of length $6n^2$, one entry per sticker, numbered face by face in the order U R F D L B and within a face left to right, top to bottom as seen from outside. For $n = 3$ this is exactly the layout `cube_nnn` describes, and `cube_group(3)` agrees with `cube_group` move for move on the twelve face turns.

The alphabet

Three axes, n layers, two directions: $6n$ moves. Half turns are *not* in it — `U2` is the word `"U U"`. This is the quarter-turn metric, and the choice matters because it is what a shortest path is counted in: the 3x3x3 face group has diameter 20 counting half turns as one move and 26 counting them as two.

Outer layers keep the letters the literature uses. The 3x3x3 slices are `M`, `E` and `S`, in their usual senses. Inner layers of larger cubes have no standard letter and are written axis-and-index, so `"1x"` is layer 1 along x.

What "solved" means

Once slices are in the alphabet the centres move, and a cube turned bodily in space is solved without its stickers being back where they started. `cube_is_colour_solved` is the test that allows for this; comparing against `cube_identity` is the stricter one that does not.

See Also

`cube_group`, `cube_moves`, `cube_nnn` for the 3x3x3 in particular

cube_orbits *Orbits of an N x N x N Cube*

Description

One row per orbit: what kind of piece it holds, how many pieces are in it, and the depths and mirror sign that name it. See cube_orbits_family for what an orbit is and why it matters.

Usage

```
cube_orbits(n)
```

Arguments

n	Side of the cube, 2 or more.

Value

A data.frame with one row per orbit and columns orbit (index), label, kind ("corner", "edge" or "centre"), stickers_per_piece, n_pieces, depth_a, depth_b and chirality (-1, 0 or 1; non-zero only for one of a mirror pair).

See Also

cube_pieces, cube_progress

Examples

```
cube_orbits(3)   # corners, edges, fixed centres
cube_orbits(4)   # centres now move, edges are wings
cube_orbits(6)   # a mirror pair of centre orbits, labelled + and -
```

cube_orbits_family *Piece Orbits of a Cube of Any Size*

Description

"Corners and edges" is a sentence about a 3x3x3. A 4x4x4 has no edges in that sense — what looks like one is two wings that never swap — and its centres move. So the vocabulary has to come from the geometry, and these functions derive it rather than tabulating it.

Pieces

A piece is a cubie: a position carrying one sticker per face it touches, and how many faces it touches is the whole classification. Three stickers is a corner, and there are always eight of them whatever the size. Two is an edge piece — the twelve edges of a 3x3x3, splitting into wings on larger cubes. One is a centre, fixed on an odd cube only at the middle of each face.

Orbits

Two pieces are in the same orbit when some sequence of moves takes one to the other, and the point of an orbit is what it forbids: no word, however long, moves a piece out of its own. A method may therefore treat each orbit separately, which is most of what solving a large cube consists of.

Orbits are computed by walking the cube move by move from one piece and collecting everywhere it can go, so they are orbits by definition rather than by a rule that might be wrong. The labels only name what the walk found. A label reads `f3:0,0,0` for the corners, `f2:0,0,k` for an edge orbit and `f1:0,j,k` for a centre orbit, where the numbers are each coordinate's distance to the nearer end of its range, sorted.

Where a class of centres holds two orbits that are mirror images of each other, the labels carry a trailing `+` or `-`. This happens on some sizes and not others — a 6x6x6 splits `f1:0,1,2` and a 5x5x5 does not — which is why the orbits are walked rather than guessed.

Every orbit holds 24 pieces except three cases: the 8 corners, and on an odd cube the 6 fixed centres and 12 middle edges, all of which sit on a symmetry axis.

See Also

cube_orbits, cube_pieces, cube_progress, cube_group

`cube_pair_edges` *Pair the Edges of a Cube of Any Size*

Description

The second stage of reduction, after the centres are built: bring the pieces of each cube edge together so the cube can be finished as a 3x3x3. This is the generic form of `pair_edges()` in `src/cube_edges.h`.

Usage

```
cube_pair_edges(
  state,
  n = NULL,
  depth = 3L,
  max_rounds = 40L,
  keep_centres = TRUE
)
```

Arguments

`state`	Integer vector of $6n^2$ stickers.
`n`	Side of the cube; inferred from `state` when `NULL`.
`depth`	Longest setup word, in outer turns. Default 3.
`max_rounds`	Give up after this many rounds. Default 40.
`keep_centres`	Reject any candidate that disturbs the centres.

Value

A list with `state` (the cube after the stage), `path` (the moves applied, as a character vector), `whole` (how many of the twelve edges ended whole), `rounds` and `solved`.

How it works

Try every algorithm under every setup word, take whatever pairs the most, repeat. The setup words are outer turns only — they carry an edge whole (measured; see `cube_edge_map_family`), so they can reposition the cube without ever undoing work. The algorithms are the six of `cube_edge_algs`. A round that cannot improve ends the stage.

A candidate is rejected outright if it disturbs the centres, so the stage can never spend what the centre stage built. That check is what `keep_centres` controls; turning it off is for diagnosis, not solving.

On the setup pool

The reference widened its pool to every composition of up to three outer turns, having measured five positions that stalled with anything shorter. `depth` is that number. Three gives 1885 words and is the default; two gives 157 and is enough to watch the stage work but not to finish reliably.

See Also

`cube_edge_algs`, `cube_edge_counts`, `cube_build_lslice`

Examples

```
set.seed(1)
s <- generate_state(group = cube_group(5), n_moves = 20)
r <- cube_pair_edges(s, depth = 2, max_rounds = 5)
r$whole
```

---

cube_piece_layout *The Piece Layout a Cube State Is Read Through*

Description

Which stickers belong to which physical piece, in the form the piece encoding needs. Built from `cube_pieces` rather than written out here: which three stickers make a corner is a fact about the cube, and one measured table beats a transcribed one.

Usage

```
cube_piece_layout(group)
```

Arguments

`group`	A `perm_group` built by `cube_group`

Value

A list describing the layout: `slots` (integer matrix, one row per piece, its sticker positions padded with 0), `n_piece`, `width` (slots per piece), `home` (which piece each sticker belongs to) and `turn` (which of its slots each sticker is)

What it is for

A cube state is a permutation of sticker positions, and handing that to a network as 54 numbers makes it discover for itself that positions 1, 37 and 48 always travel together. They do so because they are one corner. The encoding below says that outright, which is the representation DeepCubeA uses: for every piece, which slot it currently occupies and how it is turned.

Centres

Centres are dropped only when no move disturbs them. Under face turns alone they never move, so on a 3x3x3 restricted to U/R/F/D/L/B they carry no information and are left out, giving DeepCubeA's 20 pieces. Add the slice moves M, E or S and the centres do permute — each slice moves four of them — so there they are kept and the layout has all 26. The decision is taken by applying the group's own moves and looking, not from the move names.

See Also

`cube_adi_model`, `cube_pieces`

Examples

```
g <- cube_group(3, moves = c("U", "U'", "R", "R'", "F", "F'",
                             "D", "D'", "L", "L'", "B", "B'"))
lay <- cube_piece_layout(g)
lay$n_piece   # 20: centres are fixed under face turns, so they are dropped
```

`cube_pieces` *Pieces of an N x N x N Cube*

Description

One row per piece: where it sits, which orbit it belongs to, and which stickers of the state vector it carries.

Usage

```
cube_pieces(n)
```

Arguments

`n`	Side of the cube, 2 or more.

Value

A `data.frame` with one row per piece and columns `piece`, `x`, `y`, `z`, `orbit`, `label`, `n_stickers` and `stickers` — the last a comma-separated list of 1-based sticker indices.

See Also

`cube_orbits`, `cube_progress`

Examples

```
p <- cube_pieces(3)
nrow(p)                       # 26: 8 corners, 12 edges, 6 centres
head(p[p$n_stickers == 3, ])  # the corners
```

---

cube_pieces_home *Pieces Home on a Cube*

Description

The blunt summary of `cube_progress`: how many pieces are home out of how many there are.

Usage

```
cube_pieces_home(state)
```

Arguments

`state`	Integer vector of $6n^2$ stickers.

Value

A named integer vector with elements `home` and `total`.

See Also

`cube_progress`

Examples

```
cube_pieces_home(cube_identity(4))   # all of them
```

cube_predicates *Stage Predicates of a Cube State*

Description

Every stage of a solving method is "search until some predicate holds", so these predicates are the definitions of the stages. Reported together because what matters is usually which of them changed.

Usage

```
cube_predicates(state)
```

Arguments

state	Integer vector of 54 stickers

Value

Named logical vector: `cross_solved`, `slot_1` to `slot_4`, `f2l_solved`, `oll_solved`, `first_layer_solved`, `ll_cross_oriented`, `ll_edges_placed`, `ll_corners_placed`, `cube_solved`

See Also

`cube_read_state`, `cube_solve_cfop`

Examples

```
cube_predicates(cube_identity(3))   # all TRUE
```

cube_progress *How Much of Each Orbit Is Solved*

Description

A measure of progress that works at any size. One number for a whole cube says too little — the orbits are solved at different times, and a method that has finished the centres and not the wings is in a quite different position from one that has done neither.

Usage

```
cube_progress(state)
```

Arguments

state	Integer vector of $6n^2$ stickers.

Details

A piece counts as home when every sticker in its slot shows the colour that slot has on a solved cube. That is a statement about colours rather than about sticker numbers, so a cube solved relative to its centres counts as solved — which is what a real cube looks like, and what the blindfold methods mean by the word.

The size is read from the state: a vector of 96 entries is a 4x4x4 and there is nothing else it could be.

Value

A `data.frame` with one row per orbit and columns `orbit`, `label`, `kind`, `solved`, `total` and `fraction`.

See Also

`cube_orbits`, `cube_pieces_home`

Examples

```
set.seed(42)
s <- generate_state(group = cube_group(4), n_moves = 30)
cube_progress(s)
```

`cube_read_state` *Read a Cube State as Pieces*

Description

A cube state is 54 stickers, but a cuber sees 8 corners and 12 edges, each in some slot and turned some way. This is that reading, and it is what every stage predicate is built on: `cp[i]` is which corner sits in slot `i`, `co[i]` how far it is twisted, and likewise `ep`/ `eo` for edges. All indices are 0-based, as the C++ layer has them.

Usage

```
cube_read_state(state)
```

Arguments

`state`	Integer vector of 54 stickers

Details

On a solved cube every `cp`/`ep` is its own index and every `co`/`eo` is zero.

Value

List with components `cp`, `co` (length 8) and `ep`, `eo` (length 12)

See Also

cube_predicates, cube_apply_word

Examples

```
cube_read_state(cube_identity(3))
```

---

cube_reduce_best *Reduce a 4x4x4, Trying Every Starting Face*

Description

cube_reduce_cpp builds its first centre on one face and works from there. Which face that is was written into the pipeline as L; this runs it once per face and keeps the shortest reduction that actually reduces.

Usage

```
cube_reduce_best(state, faces = 0:5, verify = TRUE)
```

Arguments

state	Integer vector of 96 stickers.
faces	Which starting faces to try, 0 to 5 in the order U R F D L B. Defaults to all six.
verify	Whether to replay each path and keep only those that leave the cube reduced. On by default: a face that reports success without reducing would otherwise win by being short.

Value

The result of cube_reduce_cpp for the best face, with two components added: face, the face it started from, and tried, a data.frame of every face with columns face, found, n_moves and verified. If no face reduces the cube, the result of the first face tried is returned with found left as it came.

Why the face matters

The method does not search for a short reduction, it follows a schedule — a centre, the layer beside it, the rest shot down from the top, then the edges — and the schedule runs whatever the cube arrived like. So the cost is decided by how the scramble happens to sit relative to the face the schedule starts on, and that is a property of the pairing, not of the cube.

Measured over 200 scrambles of depth 2 to 20, six faces each, every path verified by replaying it (inst/examples/ carries the survey):

- the default face is the best of the six on 22% of cubes;
- the median cube reduces 26% shorter by picking the best face, and 30038 moves over the whole set become 19202 — 64% of the default;

- no face wins often enough to be a better default: U leads at 53 of 200 and L, the old default, still wins 28. The gain is in trying them.

The saving holds at every depth measured, from 94% at depth 2 down to about 18% at depth 20, where the default already costs some 190 moves.

What it costs

Six reductions instead of one: about 0.32 seconds against 0.053 on the machine the survey ran on. Nothing is shared between the six — each starts from the state as given.

See Also

cube_reduce_cpp, cube_solve4, cube_is_reduced

Examples

```
set.seed(42)
s <- generate_state(group = cube_group(4), n_moves = 12)

res <- cube_reduce_best(s)
res$face
res$tried
length(res$path) <= length(cube_reduce_cpp(s)$path)
```

---

cube_reduce_cpp *Reduce a 4x4x4 by the human method*

Description

Builds the centres, then pairs the wings with the algorithms speedcubers use – the setup-and-pair method, including the last-pair case and the OLL parity. The algorithms are listed in edge_algs() in src/cube_edges.h; nothing here searches for a move, it chooses among measured ones.

Usage

```
cube_reduce_cpp(state, target_face = 4L)
```

Arguments

state	Integer vector of 96 stickers.
target_face	Face, 0 to 5 in the order U R F D L B, on which the first centre and the layer beside it are built. Defaults to L.

Details

What it does not have is a fixed order. Each round takes whichever (setup, algorithm) pair leaves the most wings paired, which is where a greedy method can reach a position none of its algorithms improve. The script inst/examples/diag_pair_edges_stall.R measures how often that happens and what shape the position has.

Value

A list with `found`, `path`, `stages`, `states` (the cube after each stage) and `failure` – the reason it stopped, empty when it did not.

See Also

`cube_solve4`, which runs this and then finishes the 3x3x3 it leaves behind.

---

cube_santa *The Santa 2023 Cube Notation*

Description

The Kaggle Santa 2023 puzzles name a cube's moves `f0`, `r1`, `-d2` and so on: a letter for the axis, an index for the layer, and a leading minus for the inverse. That is the same alphabet `cube_moves` builds — one quarter turn of one layer, $6n$ of them, half turns excluded — written differently, so the two are interchangeable once the correspondence is pinned down.

What differs

Two things, and neither can be guessed from the names.

The faces are numbered in a different order. Santa's `0..5` are our `U`, `F`, `R`, `B`, `L`, `D`; ours are `U R F D L B`. Inside a face the two layouts agree exactly — row by row, left to right, seen from outside — so the relabelling moves whole faces and never disturbs a face's interior.

The senses do not line up letter for letter. `f3` is `B'` rather than `B`, `d3` is `U'`, `d1` is `1y'`. Written in axes and layers instead the irregularity disappears, which is why the correspondence is defined that way here:

Santa	axis	layer, quarter turns
fi	z	layer $n - i$, 3 turns
ri	x	layer $n - i$, 3 turns
di	y	layer $i + 1$, 1 turn

A leading `-` inverts, which is 1 turn against 3.

Which correspondence

Sixteen relabellings satisfy all $6n$ equations, because a cube may be held sixteen ways that leave this alphabet looking like itself — four turns about the vertical, a choice of which pole is up, and a mirror. They are the same correspondence seen from different angles, not a genuine ambiguity, and the one used here is the one that keeps `f0` as `F` and `r0` as `R`.

States

A Santa state is $6n^2$ colours `0..5` in their face order; ours is $6n^2$ sticker numbers. `cube_santa_state` and `cube_santa_state_out` carry a state across, and `cube_santa_perm` carries the relabelling itself.

See Also

`cube_santa_moves`, `cube_santa_state`, `cube_moves`, `cube_nnn`

---

cube_santa_group *A Cube Group in Santa 2023 Notation*

Description

The same group `cube_group` builds, with its moves named the way Santa 2023 names them. Solvers and searches work on it unchanged; only the names in the paths they return differ.

Usage

```
cube_santa_group(n)
```

Arguments

`n`	Integer, the side of the cube (at least 2)

Value

An external pointer to the group, as `cube_group` returns

See Also

`cube_santa`, `cube_group`

Examples

```
g <- cube_santa_group(3)
group_apply(g, cube_identity(3), c("f0", "-d2"))
```

cube_santa_move_names *Santa 2023 Move Names for a Cube*

Description

The $6n$ move names Santa 2023 uses for a cube of side n, in the order its puzzle files list them: `f0`, `-f0`, `f1` and so on through `f`, `r` and `d`.

Usage

```
cube_santa_move_names(n)
```

Arguments

`n`	Integer, the side of the cube (at least 2)

Value

Character vector of length $6n$

See Also

`cube_santa`, `cube_santa_moves`

Examples

```
cube_santa_move_names(2)
cube_santa_move_names(4)
```

cube_santa_moves *Translate Between Santa 2023 and Package Move Names*

Description

`cube_santa_moves` names, for each Santa move, the package move that does the same thing; `cube_moves_santa` is its inverse.

Usage

```
cube_santa_moves(n)

cube_moves_santa(n)
```

Arguments

`n`	Integer, the side of the cube (at least 2)

Details

The map is not letter for letter — `f3` is `B'` on a 4x4x4, and `d3` is `U'` — so it is derived from the geometry each time rather than tabulated. See `cube_santa` for why.

Value

Named character vector: names are the source notation, values the target one

See Also

`cube_santa`, `cube_santa_perm`

Examples

```
m <- cube_santa_moves(4)
m[c("f0", "f3", "d3", "d1")]

# and back again
cube_moves_santa(4)[c("F", "B'", "U'")]
```

---

cube_santa_path *Translate a Santa 2023 Solution Path*

Description

A Santa path is its moves joined by dots, as the submission files write them. `cube_santa_path` turns one into a character vector of package moves, and `cube_santa_path_out` turns package moves back into a Santa path.

Usage

```
cube_santa_path(path, n)

cube_santa_path_out(path, n)
```

Arguments

`path`	For `cube_santa_path`, a Santa path: one dot-separated string, or a character vector of move names. For `cube_santa_path_out`, a character vector of package moves.
`n`	Integer, the side of the cube (at least 2)

Details

Half turns have no name in either alphabet — both count in quarter turns — so a package move written `U2` becomes two Santa moves.

Value

cube_santa_path returns a character vector of package moves; cube_santa_path_out returns one dot-separated string.

See Also

cube_santa, cube_santa_moves

Examples

```
cube_santa_path("f0.-d3.r1", n = 4)
cube_santa_path_out(c("F", "U", "2x"), n = 4)
```

---

cube_santa_perm *Sticker Relabelling Between Santa 2023 and This Package*

Description

The permutation carrying a Santa 2023 sticker position to the package's own. Santa numbers the faces U F R B L D and we number them U R F D L B; within a face the two agree, so this permutation moves whole faces and nothing else.

Usage

```
cube_santa_perm(n, inverse = FALSE)
```

Arguments

n	Integer, the side of the cube (at least 2)
inverse	Logical, return the permutation the other way round

Value

Integer vector of length $6n^2$

See Also

cube_santa, cube_santa_state

Examples

```
p <- cube_santa_perm(3)
length(p)

# applying it twice with the inverse gets back where it started
identical(cube_santa_perm(3, inverse = TRUE)[p], seq_len(54L))
```

cube_santa_state *Read and Write Santa 2023 Cube States*

Description

cube_santa_state turns a Santa state — the colours 0..5, or the letters a puzzle file writes them as — into a package state. cube_santa_state_out goes back the other way.

Usage

```
cube_santa_state(state, n = NULL)

cube_santa_state_out(state, n = NULL)
```

Arguments

state	For cube_santa_state, the Santa state: an integer vector of $6n^2$ entries — colours 0..5 or distinct sticker numbers 0..6n^2-1 — or a character vector of face letters, or a single comma-separated string as the puzzle files store it. For cube_santa_state_out, a package state in either matching form.
n	Integer, the side of the cube; inferred from the length when omitted

Details

Santa writes a state in one of two ways, and both are read here. Most of its cube puzzles give a colour per sticker — 0..5, or the letters A to F — which is what cube_is_colour_solved and the solvers compare. Some give a distinct number per sticker instead, 0 to $6n^2 - 1$, which is a permutation and carries strictly more information: it says which of four identically coloured centres is which.

Both forms are relabelled the same way, because the relabelling acts on positions. Colours are additionally renumbered, since a colour is a face number and the two conventions order their faces differently; distinct sticker numbers are renumbered too, into the package's 1..6n^2.

Value

Integer vector of length $6n^2$: colours 0..5 numbered by the receiving convention's face order, or sticker numbers, matching whichever form was given. Package sticker numbers are 1-based and Santa's are 0-based.

See Also

cube_santa, cube_santa_perm

Examples

```
# a solved Santa 3x3x3 is nine of each colour in face order
s <- rep(0:5, each = 9L)
cube_santa_state(s)

# and back
identical(cube_santa_state_out(cube_santa_state(s)), s)

# the permutation form: Santa's solved state is 0..53, ours is 1..54
identical(cube_santa_state(0:53), seq_len(54L))
```

---

cube_slice_cells_cpp *The Layer Beside a Face, as the 4x4x4 Solver Sees It*

Description

The twelve centre cells of the layer lying against one face: that face's own four, together with the column each of the four faces around it presents to it. This is what step 2 of the reduction builds and what step 3 must not disturb.

Usage

```
cube_slice_cells_cpp(face)
```

Arguments

face	Face index, 0 to 5, in the order U R F D L B.

Value

A `data.frame` of twelve rows with columns `face` and `slot`.

Derived from the slice table

`src/cube_centres.h` used to write two of these out by hand — one for L, which step 2 builds, and one for D, where the rotation puts it. Both are read off the measured slice map instead: the layer of a face is that face plus whatever the slice lying beside it carries, and which of the two slices on the axis is the near one is settled by turning a solved cube and seeing which leaves the layer whole.

The slots are numbered as `centre_slots_of` in the C++ numbers them, which is not the numbering `cube_centre_structure` uses — that one runs along the face-turn cycle. The two tables are not comparable cell by cell.

See Also

`cube_reduce_cpp`, `cube_slice_map`

Examples

```
# the layer beside L: all of L, and a column of each face around it
cube_slice_cells_cpp(4)

# every face has one, and each is twelve cells
vapply(0:5, function(f) nrow(cube_slice_cells_cpp(f)), integer(1))
```

`cube_slice_map` *Where a Slice Turn Sends Each Centre*

Description

The generic form of the slice table that `src/cube_centres.h` writes out by hand for a 4x4x4: one row per centre sticker the turn moves, saying which (face, orbit, slot) it leaves and which it arrives at.

Usage

```
cube_slice_map(n, move, structure = NULL)
```

Arguments

`n`	Side of the cube, 2 or more.
`move`	A move name from `cube_move_names`, such as `"1x"` or `"U"`.
`structure`	Optionally the result of `cube_centre_structure` for the same `n`, to avoid recomputing it when mapping many moves.

Value

A `data.frame` with columns `from_face`, `from_orbit`, `from_slot`, `to_face`, `to_orbit` and `to_slot`, with no rows if the move leaves every centre alone.

See Also

`cube_centre_structure`, `cube_centre_map_family`

Examples

```
# a 4x4x4 slice moves eight centres, in one orbit
m <- cube_slice_map(4, "1x")
nrow(m)

# no slice ever mixes two orbits, at any size
m5 <- cube_slice_map(5, "1x")
all(m5$from_orbit == m5$to_orbit)
```

`cube_solve_centres` *Solve the Centres of a 4x4x4 Cube*

Description

The first stage of reduction. On a 3x3x3 a centre is one sticker and cannot move; from 4x4x4 up each face carries four centre pieces, all twenty-four in a single orbit, and they have to be gathered before the cube can be treated as a 3x3x3 at all.

Usage

```
cube_solve_centres(state)
```

Arguments

`state`	Integer vector of 96 stickers, a reachable cube state — from `generate_state` with `group = cube_group(4)`.

Details

The method is Stefan Pochmann's layer-by-layer centres — the same author as the M2 method in `cube_solve_m2`. One centre is built, turned to the left, and the layer beside it is filled; then that layer goes to the bottom and the remaining pieces are shot down from the top one at a time. Two things are computed rather than searched for: which move carries a given piece where, and how far to turn U first so the shot has something to fire.

Whatever the shooting cannot reach is finished by commutators — a swap between two faces, or a 3-cycle among three — with the cube turned to meet each one rather than a variant of each kept per orientation. A 4-cycle is not a separate case: it is a 3-cycle followed by a swap.

Value

List with components:

`path`	Character vector of moves ("R", "1x'", ...)
`found`	Logical, whether all six centres ended built
`stages`	data.frame of stages: `name`, `detail`, `n_moves`. The moves themselves are on the `"moves"` attribute, one character vector per row.
`states`	List of integer vectors, the cube after each stage

See Also

`cube_group`, `generate_state`, `cube_is_colour_solved`

Examples

```
set.seed(42)
s <- generate_state(group = cube_group(4), n_moves = 40)

res <- cube_solve_centres(s)
res$found
length(res$path)
```

`cube_solve_cfop` *Solve a 3x3x3 Cube by CFOP*

Description

Cross, then the four first-two-layer pairs, then orient the last layer, then permute it. The cross and the pairs are searched for optimally; the last layer is looked up in the standard tables of 57 OLL and 21 PLL cases. That division is the method's own: the first half is intuitive and the second is memorised.

Usage

```
cube_solve_cfop(state, cross_depth = 8L, slot_depth = 11L)
```

Arguments

`state`	Integer vector of 54 stickers, a reachable cube state — from `generate_state` with `group = cube_group(3)`.
`cross_depth`	Integer, search depth for the cross (default 8, the standard bound: no cross is further than 8 quarter turns).
`slot_depth`	Integer, search depth for one F2L pair (default 11).

Details

Each stage is solved optimally on its own and the total is still around three times optimal, because a stage that is shortest in isolation usually leaves the next one worse off. That gap is the price of a method a person can hold in their head. Move counts are quarter turns, so they run larger than the counts quoted in speedcubing, which count a half turn as one.

Value

List with components:

`path`	Character vector of moves ("R", "U'", ...)
`found`	Logical, whether the cube ended solved
`stages`	data.frame of stages: `name`, `detail` (the case name, where there is one), `n_moves`. The moves themselves are on the `"moves"` attribute, one character vector per row.
`states`	List of integer vectors, the cube after each stage

See Also

cube_solve_lbl, cube_group, generate_state

Examples

```
set.seed(42)
s <- generate_state(group = cube_group(3), n_moves = 20)

res <- cube_solve_cfop(s)
res$found
length(res$path)
res$stages
```

---

cube_solve_lbl *Solve a 3x3x3 Cube Layer by Layer*

Description

The method a beginner is taught: cross on D, the four bottom corners, the four middle edges, then the last layer in four steps — make the yellow cross, place its edges, place the corners, twist them. Each last-layer step is a small algorithm applied over and over with a U turn between, which is why the method is easy to learn and long to execute.

Usage

```
cube_solve_lbl(state, cross_depth = 8L, corner_depth = 9L, edge_depth = 11L)
```

Arguments

state	Integer vector of 54 stickers, a reachable cube state — from generate_state with group = cube_group(3).
cross_depth	Integer, search depth for the cross (default 8)
corner_depth	Integer, search depth per first-layer corner (default 9)
edge_depth	Integer, search depth per middle edge (default 11)

Details

It shares the cross and the last layer with cube_solve_cfop. The middle is where it is slower and simpler: the bottom layer is finished outright, and only then do the middle edges go in one at a time. Expect roughly twice the moves of CFOP.

Value

List with the same components as cube_solve_cfop: path, found, stages, states.

See Also

cube_solve_cfop, cube_group, generate_state

Examples

```
set.seed(42)
s <- generate_state(group = cube_group(3), n_moves = 20)

res <- cube_solve_lbl(s)
res$found
length(res$path)
```

---

cube_solve_m2 *Solve a 3x3x3 Cube by M2*

Description

Old Pochmann's method with a cheaper edge step, and the usual next thing a blindfold solver learns. There an edge was placed by a whole PLL wrapped in setup moves; here the buffer is DF and the swap is M2, which is two moves. The corners are unchanged.

Usage

```
cube_solve_m2(state)
```

Arguments

state	Integer vector of 54 stickers, a reachable cube state — from generate_state with group = cube_group(3).

Details

M2 is not a clean swap, and the method is arranged around that rather than avoiding it. It turns the middle slice a half turn, so it moves the centres and exchanges the UF/DB edge as well. Those two get their own algorithms; edges left facing the wrong way are set aside while the chain runs and turned afterwards in one orientation phase; and whether the parity fix is needed is decided by reading the edge permutation rather than by counting swaps, because a chain that breaks into a new cycle spends turns the tally does not see.

The cube comes back solved with the centres sometimes rotated among themselves — "solved relative to the centres", which is what the method means by solved and what a real cube looks like.

Expect roughly two thirds the moves of cube_solve_old_pochmann, with the saving all on the edges.

Value

List with the same components as cube_solve_cfop: path, found, stages, states.

See Also

cube_solve_old_pochmann, cube_solve_cfop, cube_solve_lbl, cube_group

Examples

```
set.seed(42)
s <- generate_state(group = cube_group(3), n_moves = 20)

res <- cube_solve_m2(s)
res$found
length(res$path)
```

---

cube_solve_old_pochmann

Solve a 3x3x3 Cube by Old Pochmann

Description

The blindfolded method, and it is built on a different principle from the other two. CFOP and layer by layer both work by looking: a stage ends when the cuber can see that it has. Old Pochmann never looks at the cube after the start, so it repeats one step whose shape is always the same — swap whatever is in the buffer with one chosen piece, disturbing nothing else. Follow it and the pieces come home in a chain.

Usage

```
cube_solve_old_pochmann(state)
```

Arguments

state	Integer vector of 54 stickers, a reachable cube state — from generate_state with group = cube_group(3).

Details

Nothing here is searched. Each piece is placed by a conjugate: setup moves that bring the target where the algorithm can reach it, one memorised algorithm, then the setup moves undone. The algorithms are ordinary PLLs, because "swap two edges and two corners" is exactly what a T-perm does, and the second swap is not incidental: a single swap is an odd permutation and the cube group has none. Those extra swaps cancel in pairs, and when their count is odd one is left over — that is the parity step, between the edges and the corners.

Expect around twice the moves of layer by layer. The cost of never looking is that no move ever does two things at once.

Value

List with the same components as cube_solve_cfop: path, found, stages, states. Each stage's detail names the target sticker by its standard letter and the algorithm used.

See Also

cube_solve_cfop, cube_solve_lbl, cube_group, generate_state

Examples

```
set.seed(42)
s <- generate_state(group = cube_group(3), n_moves = 20)

res <- cube_solve_old_pochmann(s)
res$found
length(res$path)
```

---

cube_solve4 *Solve a 4x4x4 Cube by Reduction*

Description

Reduction is the standard way to solve an even cube: build the centres so each face acts as one, pair the edges so each acts as one, and the cube is then a 3x3x3 with fat pieces — finished by cube_solve_cfop.

Usage

```
cube_solve4(
  state,
  method = c("cfop", "kociemba", "lbl", "m2", "pochmann"),
  orientations = 1L,
  start_face = c("full", "reduction", "fixed")
)
```

Arguments

state	Integer vector of 96 stickers, a reachable cube state — from generate_state with group = cube_group(4).
method	Which 3x3x3 solver finishes the reduced cube. All five reach a solved cube and differ in length and time; measured on 3x3x3 states, "kociemba" averages about 40 moves in 0.9 s, "cfop" 103 in 0.04 s, "lbl" 169 in 0.06 s, "m2" 278 in 0.09 s and "pochmann" 433 in 0.14 s. The reduction in front of them is the same either way, so the difference in the 4x4x4 total is the difference above. Parity is detected with CFOP whichever is chosen, because refusing a state is how the parity cases are found at all, and only CFOP knows the 57 OLL and

21 PLL cases well enough to refuse. The chosen solver sees the cube after any repair.

`orientations` How many ways of holding the cube to try, keeping the shortest answer. One — the default — solves the cube as it arrives and is what this function has always done.

The reduction works the faces in a fixed order, and a scramble leaves the cube in whatever orientation it leaves it in, so that order lands well or badly by chance. Measured over the 24 orientations of one cube, the reduction returned words from 139 moves to 312 — the same cube throughout, since a rotation moves no piece relative to another, so the whole of that range is the view rather than the position.

Trying a few and keeping the least is therefore not a search for a better answer but for a better vantage. Measured over twenty cubes, counting the turn and its undoing as the moves they are, and passing each result through `short_path_bfs` as a caller would:

	solver	shipped	seconds
1 orientation	231	203	3.7
8	205	181	4.4
24	191	172	7.0

Eight is the bargain: eleven per cent shorter for a fifth more time. The last four per cent cost four times as much again — the solve itself is cheap and it is the shortener that spends the seconds, and a shorter path is quicker to shorten, which pays part of the sweep back.

This multiplies with `start_face`: each orientation tried is solved from whichever face that setting picks.

`start_face` How to choose the face the first centre is built on.

`"full"`, the default, solves from all six and keeps the shortest solution. `"reduction"` keeps the face that reduces shortest, which is cheaper but optimises the stage rather than the answer. `"fixed"` is the single face the pipeline was written to, and what this function did before the choice existed.

Measured over 40 scrambles of depth 2 to 20, total moves against `"fixed"`, and the seconds a cube costs:

	total	of fixed	seconds
fixed	9794	100%	0.06
reduction	7396	76%	0.32
full	6064	62%	1.70

`"full"` is never worse than `"fixed"` — the fixed face is one of the six it compares — while `"reduction"` was worse on 4 of the 40, once by 152 moves. The two agree on about half the cubes and diverge most from depth 6 up, where solutions are longest.

The default stays at one so that existing callers keep the time they expect; eight is a thing to ask for, not to be given. And the eight is measured on a 4x4x4 with these twenty cubes — the shape of the curve on another size, or on a differently distributed set of cubes, is not known, and picking the number again there means measuring it again.

Details

Two things happen on a 4x4x4 that cannot happen on a 3x3x3, and both are handled here. The centres and edge pairs are not fixed relative to one another, so the first two stages are searches over the cube's own geometry rather than table lookups. And reduction can leave the last layer in a state no 3x3x3 reaches — OLL and PLL parity — which the 3x3x3 tables have no entry for; those are repaired with algorithms that turn inner layers, before the cube is handed on.

Expect roughly 300 quarter turns. The method is layer by layer throughout and makes no attempt at brevity; a search-based solver reaches the same cube in a fraction of the moves and a great deal more time.

Value

List with components:

`path`	Character vector of moves
`found`	Logical, whether the cube ended solved
`stages`	data.frame of stages: `name`, `detail`, `n_moves`
`states`	List of integer vectors, the cube after each stage

What the reduction costs, and why

The length is a property of the method rather than of the position, and the clearest case measured is this. A cube eight moves from solved — scrambled by `R' D U U' 2z B 1x' 1z'` — was reduced in 204 moves. Walking the inverse scramble back shows a reduced state four moves away: the fourth move of the way home already satisfies `cube_is_reduced`. The method built one in two hundred and four while one lay in four.

It is not a defect of the implementation. The reduction does not look for the NEAREST reduced state; it builds its own by a fixed schedule — one centre, then the layer beside it, then the top slice emptied, then the edge pairs — and every stage runs whatever the cube arrived like. Following one such solution move by move, 74% of the moves changed neither the count of centres home nor the count of edges paired. The edge stage spent 107 moves pairing twelve edges, nine apiece, with the centres flickering between four and eight all the way: take them apart, pair an edge, put them back.

That is also why no cheap test predicts the price. The sum of missing centres, their arrangement over the faces, and the number of unpaired edges were each measured against the length over 420 cubes, and none separates a cheap cube from an expensive one — because there is little to separate. Only the threshold shows: while a cube is still reduced the stage does almost nothing (three to eight moves), and once it is not, the schedule runs in full at a price that barely depends on how far from solved it is.

Searching for a nearby reduced state instead does not rescue this. Reduced is a weak condition and countless states satisfy it, but iterative deepening still walks the whole tree to each depth before the next, so the many goals buy nothing — measured, a fourteen-move scramble did not finish inside six minutes at depth six. Using the gap would want a prune table over the distance to reduced, which is what `cube_kociemba` has for the 3x3x3 and this does not.

What does help is choosing where the schedule starts. If the price turns on how the scramble sits relative to the face the first centre is built on, then that face is worth picking rather than fixing, and `start_face` below is how it is picked. Measured over 200 scrambles, the face that had been written

into the pipeline is the best of the six on 22% of cubes; the reduction stage alone falls to 64% of its length when all six are tried.

Picking by the reduction is not the same as picking by the answer, though, and the difference is not small. A short reduction can leave the cube in a parity a longer one avoids, and the fifty moves that costs outweigh what it saved. Over 40 scrambles the two criteria chose differently on 48% of cubes, the reduction criterion losing 70 moves on average and 152 at worst — and on four of the forty it produced a solution LONGER than the fixed face would have. Choosing on the finished solution cannot do that, since the fixed face is one of the six it compares.

`orientations` below is the same effect reached from the other side: the same cube held differently reduces in anything from 139 moves to 312, and that range is the view rather than the position.

See Also

`cube_solve_centres`, `cube_solve_cfop`, `cube_group`

Examples

```
set.seed(42)
s <- generate_state(group = cube_group(4), n_moves = 40)

res <- cube_solve4(s)
res$found
length(res$path)
```

`cube_solve4_cascade` *Solve a 4x4x4 by reduction, falling back when the search does not pay off*

Description

Phase 3 of the reduction is usually cheap when it succeeds and expensive when it fails, and the two are told apart only afterwards. So the leash is a ladder rather than a single length: every orientation is tried at a small budget before any is tried at a large one. See `budget_steps`.

Usage

```
cube_solve4_cascade(
  state,
  node_budget = 2e+05,
  prep_budget = 5e+06,
  orientations = c("", "1y", "1x", "1z"),
  max_orientations = 4L,
  phase2_solutions = 4L,
  max_candidates = NA_integer_,
  shuffle_candidates = FALSE,
  workers = 1L,
```

```
  budget_steps = c(0.1, 0.3),
  exact_centres = TRUE,
  verbose = FALSE,
  progress_every = 0
)
```

Arguments

`state`	Integer vector of 96 stickers.
`node_budget`	The largest number of nodes phase 3 may spend on one orientation. It is the top of the ladder rather than the amount every attempt pays – see `budget_steps`. It bounds phase 3 only; phases 1 and 2 have `prep_budget`.
`prep_budget`	Nodes phases 1 and 2 may spend on one orientation, spent whole rather than divided by `budget_steps` as `node_budget` is. Measured over thirty cubes: 116 of 120 phase-2 searches finished inside 3.3 million nodes, the other four took 17.3 to 30.0 million and were the slow cubes. 2e5 is too tight – it cuts off ordinary orientations.
`orientations`	Whole-cube rotations to try, as move words. Each is applied before the reduction and undone after, which changes nothing about the cube but everything about which goal the phases aim at.
`max_orientations`	How many orientations to try before falling back. The default of 4 is a time bound, not a quality one – trying all 24 costs twenty-four times as long for the cubes that end up falling back anyway.
`phase2_solutions`	How many phase-2 solutions to take from each orientation. Each is a different state to start phase 3 from, and they are swept alongside the orientations, so this multiplies the candidates rather than replacing them. The cheap end of the same idea the orientations serve: measured on four cubes, all four handovers were distinct on every cube and the whole set cost 0.03 to 0.24 s, against the tens of seconds a failed phase 3 costs. The candidates differ in cost by factors – on one cube the second solution took 2.9 s where the first exhausted its budget in 43.5 – and the sweep's job is to reach the cheap one first. They do not replace the orientations: on one cube all four phase-2 solutions exhausted the budget while the orientation sweep solved it. Set to 1 for the old behaviour: one candidate per orientation.
`max_candidates`	How many candidates to try before giving the cube to `cube_solve4` instead. `NA`, the default, tries all of them. The dial between a short answer and a quick one. `cube_solve4` needs no search and answers in a fraction of a second at 186 to 444 moves; the reduction searches for shorter and pays the full budget for every candidate that fails. Measured on one cube: sixteen candidates, 355 s, 146 moves. A cap of 4 with `phase2_solutions = 4` means one orientation's worth of tries, then the fast answer.
`shuffle_candidates`	Try the candidates in random order.

The build order is arbitrary, not a ranking: no cheap property was found that predicts a candidate's cost. Shuffling matters when the sweep is capped – a fixed arbitrary order makes every cube's luck depend on the same accident, where a random one spreads it.

`workers` How many candidates to search at once. The candidates at a rung are independent – each is a phase-3 search from its own state, sharing nothing – so they parallelise without any coordination.

What it saves is bounded by where the winner sits: a cube won on the first candidate gains nothing, one won on the sixteenth gains nearly the full divisor. Read `candidate` in the return value before choosing.

The caution is the prune table, not the CPU: it is a process-wide 256MB table, shared copy-on-write while nobody writes, and a search deepening past its fill depth *does* write, so N workers then cost N tables. Use several workers only with a prefilled table. Windows has no fork and degrades to sequential.

`budget_steps` The rungs of that ladder, as fractions of `node_budget`. Every orientation is tried at the first rung before any of them is tried at the second, so a cube that phase 3 finds cheaply is found cheaply whichever orientation it needed.

Why a ladder: failure costs the full budget by definition, since `exhausted` is a statement about the leash and not about the cube. One cube exhausted 5e7 twice, 84 s, before being found on a third orientation in 25.9 s. Sweeping every orientation low first costs a fraction of one failure high, and nothing is wasted – the search restarts at each rung.

Where the rungs sit matters more than how many. A first rung of 2e6 was worse than no ladder for a cube that succeeded at 2.4 million nodes: all four orientations exhausted it first, 6.2 s to 12.7 s. Hence 0.1.

The rung at 1.0 was removed. Over thirty cubes it cost roughly 920 s of the run and yielded three cubes worth having (152, 134, 152 moves); of the ten it solved, eight came back at 294 to 312 moves, which `cube_solve4` matches for free at 258 to 396. Set `c(0.1, 0.3, 1)` to buy those three back at roughly triple the median time, or `1` for one rung at the full budget.

The rung at 0.3 has the same weakness in smaller form, and it is not yet measured whether it should go. Over thirty cubes it carried seven, but three of those returned 294 to 316 moves – no shorter than the fallback – and every one of its successes landed between 5.3 and 12.3 million nodes, so the rung is never spent to its limit by a winner.

`exact_centres` Whether phase 3 uses the exact centre table. Measured: it removes the false zero bounds the hash table returns and solved one of six cubes that were failing, so it is on by default here.

`verbose` Print what each stage did, and say which orientation and budget is about to be tried before it is tried.

`progress_every` Nodes between progress lines from inside phase 3, or `0` for silence. Phase 3 is the only stage that can spend tens of seconds in one call – measured at 43 s for a single exhausted orientation – and with nothing printed it is indistinguishable from a hang. A value around 1e6 gives a line roughly every second at the rate that was measured.

Details

When no orientation reduces the cube at any rung, `cube_solve4` finishes the job. That solver always succeeds and its solutions are long – 186 to 444 moves on six random cubes – so it is the floor rather than the goal.

The fallback is not a failure mode to be minimised away. A reduction that does not finish inside its budget has not proved the cube unreachable – the outcome is `exhausted`, which means the budget ran out, and phase 3's true distance is simply further than the leash allowed. What the cascade buys is a bounded wait: an answer of some length always, rather than an optimal one sometimes.

Value

A list with `path` (the moves), `method` (`"reduce"`, `"solve4"`, `"already-solved"` or `"unsolved"`), `solved` (the path was replayed and checked, not assumed), `orientation` (the rotation that worked, empty for none), `candidate` (which of that orientation's phase-2 solutions it was, as `"1y#4"` – the rotation alone does not say, and with several solutions per orientation the difference is most of the time a slow cube spends), `n_moves`, `seconds`, `attempts`, and `ops`.

`ops` is a data.frame with one row per operation performed, in the order performed: `op` (`"phase 1"`, `"phase 2"`, `"phase 3"`, `"tail solve4"`, `"verify"` and so on), `secs`, `detail` (the orientation, and for phase 3 the rung), `nodes` and `outcome`. Unlike `attempts` it covers the attempts that failed as well as the one that worked, which is where most of a slow cube's seconds are: its rows sum to `seconds` up to a final `"unaccounted"` row, so nothing is left implicit.

Examples

```
## Not run:
set.seed(8)
mv <- cube_moves(4); names(mv) <- cube_move_names(4)
s <- cube_identity(4)
for (m in sample(cube_move_names(4), 6, replace = TRUE)) s <- s[mv[[m]]]
r <- cube_solve4_cascade(s)
r$method

## End(Not run)
```

---

cube_wide_turn *Wide Turns of an N x N x N Cube*

Description

A wide turn is a face together with the inner layer behind it, turned as one slab — `Rw`, `Uw`, `Dw` in the notation the published edge algorithms are written in. This gives the pair of package moves that make one up, at any size.

Usage

```
cube_wide_turn(n, face, prime = FALSE)
```

Arguments

n	Side of the cube, 3 or more.
face	One of "U", "R", "F", "D", "L", "B".
prime	TRUE for the anticlockwise turn.

Value

A character vector of package moves.

Why this is measured

src/cube_edges.h carries the expansions as constants: Dw -> D 1y\', Uw -> U 2y, Rw -> R 2x. Two things in those are 4x4x4 facts — which numbered slice lies behind a face, and nothing else. Measured across n = 4, 5, 6: a slice k<axis> turns layer k, and the faces sit at layers 0 and n - 1, so the layer behind a face is 1 for D, L, B and n - 2 for U, R, F. At n = 4 that returns the three constants above.

The direction is not free either, and edge-wholeness cannot decide it: both directions leave 8 edges whole, and both have order 4, because a face and a slice on the same axis commute. What separates them is whether the slab turns as one rigid rotation. Measured by asking whether every piece the composite moves undergoes the same rotation in the plane perpendicular to the axis: U, R, F take the slice straight and D, L, B take it inverted — uniform at n = 4 and n = 5 alike, and agreeing with the three constants.

See Also

cube_edge_algs, cube_moves

Examples

```
cube_wide_turn(4, "R")        # R 2x  -- the Rw of the sources
cube_wide_turn(5, "R")        # R 3x  -- the same turn on a bigger cube
cube_wide_turn(4, "D")        # D 1y\' -- low faces take the slice inverted
```

---

cube_word_order *Order of a Word of Moves*

Description

How many times a word must be repeated before the cube returns to where it started. A quarter turn has order 4 and a half turn order 2; a word that is not what its name says will have some other order, which is what makes this a check on the alphabet rather than a curiosity.

Usage

```
cube_word_order(word, max_order = 1260L)
```

Arguments

word	Character, moves in standard notation
max_order	Integer, give up after this many repetitions (default 1260, the largest order in the cube group)

Value

Integer, the order, or -1 if none was found within max_order

See Also

cube_apply_word

Examples

```
cube_word_order("R")        # 4
cube_word_order("R2")       # 2
cube_word_order("R U R' U'")  # 6
```

---

cycle_shortcut *Shorten a Path by Cutting Across Cycles*

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 = NULL,
  n_points = 20L,
  moves = NULL,
  group = NULL,
  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, in the group's own spelling
start_state	Integer vector, the state the path starts from
k	Integer, length of the reverse-prefix (flipper) operation. Ignored when group is given.
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	Operations the combos are drawn from, naming a subset of the group's alphabet (default: all of it)
group	A perm_group. When NULL (default) the arguments describe TopSpin.
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`.

`cube4_model` Moves still to play on a 4x4x4, read off a trained model rather than counted. The other methods score how far two states are *as arrays*, which on a cube says little about how far they are as *cubes*: two states one quarter turn apart can differ in most of their stickers. This one asks a model that was trained to predict exactly that number. Like `human` it scores against the solved cube and ignores `target`, so it belongs to searches heading for the solved state. See `find_best_match_cube4_model`.

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_cube4_model

Score Candidate States With a Trained 4x4x4 Model

Description

Distance is the model's own answer to "how many quarter turns from solved": it scores the 24 moves of a state and the smallest of them is taken as the state's value. The whole candidate set goes through in one call, because the model batches and a per-row call would cost far more than the scoring.

Usage

```
find_best_match_cube4_model(states, target, k)
```

Arguments

states	Integer matrix, one candidate 4x4x4 state per row, in sticker numbers as the rest of the package uses
target	Integer vector, ignored; accepted for interface uniformity
k	Integer, ignored; accepted for interface uniformity

Details

Scoring is against the solved cube and target is ignored, the same arrangement find_best_match_human uses and for the same reason: the model was trained towards one particular state and cannot be asked about distance to an arbitrary one. Use it for searches heading for the solved cube; on a search heading elsewhere the numbers do not describe the pair being compared.

The value is a prediction, not a count. It is close enough to guide a choice between candidates – which is all bridge selection needs – but it carries no guarantee, and a search that needs an exact answer near the solved cube should get it from the group, not from here.

The model is read once and kept for the session. Set CUBE4_ARCHIVE to the directory holding model/model.pth; ggmlR must be installed.

Registered as distance method "cube4_model"; see distance_methods.

Value

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

See Also

distance_methods, find_best_match_human

---

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 = NULL,
  combo_length,
  n_samples,
  n_top,
  start_state,
  k = NULL,
  group = NULL,
  sort_by = c("longest", "most_unique"),
  max_moves = 10000000L
)
```

Arguments

moves	Allowed operations, naming a subset of the group's alphabet (default: all of it). For TopSpin, 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. Ignored when group is given.
group	A perm_group. When NULL (default) the arguments describe TopSpin.
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).
max_moves	Integer, cap on how far a word is spun while measuring its cycle (default 1e7)

Value

Data frame with columns:

combo_number	Integer sequence number
combination	The operation sequence, written in the group's own move names, space separated
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 = NULL,
  group = NULL,
  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. Ignored when group is given.
group	A perm_group. When NULL (default) the arguments describe TopSpin.
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` *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 = NULL,
  moves = NULL,
  group = NULL,
  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"),
  max_scored = NULL,
  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. Ignored when `group` is given.
`moves`	Allowed operations, naming a subset of the group's alphabet (default: all of it)
`group`	A `perm_group`. When `NULL` (default) the arguments describe TopSpin. Only `distance_method = "manhattan"` is defined for other groups: "human" and "breakpoints" score closeness on a ring, which a cube has no counterpart for.
`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").
max_scored	Integer, at most this many candidates are scored when choosing a bridge; a random sample is taken when a cycle offers more. NULL (default) scores every candidate. Only methods that pay per state are affected – "manhattan" and "human" score in C++ and ignore it. A cycle's candidates lie along one sequence rather than spread across the space, so a sample of them ranks nearly as well as all of them; with "cube4_model", where a state costs milliseconds, this is the difference between seconds and minutes per cycle.
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 by walking away from the identity with random moves. The walk stays inside the group, so the result is always reachable from where it started — which a random permutation of `1:n` would not be. On the cube that distinction is the whole point: of the $54!$ arrangements of 54 stickers only about 4.3×10^{19} are positions a cube can actually reach.

Usage

```
generate_state(
  n = NULL,
  k = NULL,
  n_moves = 25L,
  moves = NULL,
  group = NULL,
  max_attempts = 100L
)
```

Arguments

`n`	Integer, the size of the permutation. Optional when `group` is given, since the group fixes it; required otherwise.
`k`	Integer, parameter for reverse_prefix operation (TopSpin only)
`n_moves`	Integer, number of random operations to apply (default 25)
`moves`	Moves to draw from. With a `group`, a subset of its alphabet (default all of it); without one, TopSpin's operations (default `c("1", "2", "3")`).
`group`	A `perm_group` to walk in. When absent the arguments describe TopSpin.
`max_attempts`	Integer, maximum attempts to generate a non-identity state (default 100)

Details

Given a `group`, moves are drawn from that group's alphabet. Without one the arguments describe TopSpin, as they always have, and `moves` names its operations `"1"`/`"2"`/`"3"`.

Value

Integer vector representing a reachable permutation state

See Also

cube_group, for the cube's own moves and also reports the moves it used

Examples

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

# in an arbitrary group: n comes from the group
g <- cube_group(3)
s <- generate_state(group = g, n_moves = 20)
length(s)

# the centres of a real cube never move, and this state is a real one
all(s[c(5, 14, 23, 32, 41, 50)] == c(5, 14, 23, 32, 41, 50))
```

---

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)
```

`generate_walk` *Random Walks From the Solved State, With the Moves They Took*

Description

Walks away from the identity by random moves and reports the word it walked, which is what separates this from `generate_state`. That function hands back a state and forgets the route; this one hands back the route as well, and the route is a way home — `invert_path` of it solves the state.

Usage

```
generate_walk(
  group,
  n = 1L,
  n_moves = 10L,
  exact = TRUE,
  no_undo = TRUE,
  pivot = FALSE,
  k_min = 2L,
  no_cancel = FALSE
)
```

Arguments

`group`	A `perm_group` to walk in
`n`	Number of walks (default 1)
`n_moves`	Length of each walk, or the longest one when `exact = FALSE` (default 10)
`exact`	Walk `n_moves` exactly (default `TRUE`), or draw each length uniformly from 1 to `n_moves`
`no_undo`	Refuse a move that immediately undoes the previous one (default `TRUE`)
`pivot`	Report a point partway along each walk rather than its end (default `FALSE`)
`k_min`	Shallowest cut to draw when `pivot = TRUE` (default 2). A cut of 1 puts the pivot one move from solved, where every method is right and the row says nothing.
`no_cancel`	Refuse a move that cancels against the tail of the walk through moves it commutes with (default `FALSE`)

Details

That matters wherever a solver is measured rather than merely run. Asking where an estimator ranked the helping move, or at which step a search dropped the trail, needs a state whose answer is known in advance, and the only cheap way to know one is to have walked there.

Two switches, because generating states for training and generating them for measurement pull opposite ways:

`exact` `TRUE` (default) walks `n_moves` every time, which is what measurement wants: a result is then about that depth and not about a mixture. `FALSE` draws a length uniformly from 1 to `n_moves`, the sampling training asks for, spreading accuracy outward from the goal without weighting the loss.

`no_undo` `TRUE` (default) refuses a move that undoes the one before it. `generate_state` allows those, so its `n_moves` is an upper bound on the distance — a drawn `U U\'` quietly costs two moves and travels nowhere. Refusing them makes the label mean more. Longer cycles are left alone: four quarter turns of one face also come home, and refusing every way back would mean searching the group rather than walking it.

`pivot` `TRUE` reports a point partway along each walk instead of its end, with the two moves that meet there. A cut is drawn uniformly from `k_min` to `n_moves - 1` — before the walk, so that every depth is equally represented — and `states` then holds that point, `depth` its distance label, and `prev_move` / `next_move` name the move back toward the identity and the move the walk went on to make. Those two are the only moves whose value is known at that state — `depth - 1` and `depth + 1` — which is what a Q-model is trained on. Default `FALSE`.

`no_cancel` `TRUE` refuses a move that cancels against the tail of the walk through moves it commutes with. On the cube U, D and E commute, so `U D U\'` walks three moves and travels one, and U has order 4, so `U U U` is `U\'`; `no_undo` lets both through. Commutation and order are read off the permutations, so this works for any group. It includes what `no_undo` refuses. The length is still an upper bound, only a tighter one. Default `FALSE`.

Even with `no_undo` the length stays an upper bound on the true distance to solved: a shorter route may exist that this walk did not take. A solver compared against `solution` is therefore never credited with a shortcut the reference did not have, which is the safe direction to err in.

Value

A list with

`states` integer matrix, `n` rows, one walk endpoint per row — or the pivot point when `pivot = TRUE`

`depth` integer vector, the length of each walk — or the pivot's distance label when `pivot = TRUE`

`moves` list of `n` character vectors, the word each walk took

`solution` list of `n` character vectors, each the inverse of the corresponding word — a path from that state to solved. `NULL` when `pivot = TRUE`, where the word describes the whole walk and not the reported state.

`prev_move`, `next_move` character vectors, present only when `pivot = TRUE`: the move back toward the identity and the move away from it

See Also

`generate_state` when the route is not needed, `invert_path`, `cube_group`

Examples

```
g <- cube_group(3)
w <- generate_walk(g, n = 2, n_moves = 6)
dim(w$states)
w$moves[[1]]
```

```
# the reported solution really does solve the state
s <- group_apply(g, w$states[1, ], w$solution[[1]])
identical(s, group_identity(g))

# for training: a spread of depths rather than one
w <- generate_walk(g, n = 5, n_moves = 10, exact = FALSE)
w$depth

# for a Q-model: a midpoint and the two moves whose value is known there
p <- generate_walk(g, n = 3, n_moves = 8, exact = FALSE, pivot = TRUE)
data.frame(depth = p$depth, back = p$prev_move, away = p$next_move)
```

`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 = NULL,
  group = NULL,
  max_moves = 10000000L
)
```

Arguments

start_state	Integer vector, the initial permutation state
allowed_positions	Sequence of operations to repeat, in the group's own move names
k	Integer, parameter for reverse operations. Ignored when group is given.
group	A perm_group. When NULL (default) the arguments describe TopSpin.
max_moves	Integer, cap on how far the word is spun before giving up (default 1e7)

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")

# the order of the sexy move, the long way round
g <- cube_group(3)
get_reachable_states_light(group_identity(g), "R U R' U'", group = g)
```

`group_apply` *Apply a Move Sequence Through a Group*

Description

Applies moves left to right. Moves may be named or given as indices into the group's alphabet.

Usage

```
group_apply(group, state, moves)
```

Arguments

`group`	A `perm_group`
`state`	Integer vector, the state to act on
`moves`	Move sequence: names, indices, or a single space-separated string

Value

Integer vector, the state after the moves

See Also

`group_compose`, `group_inverse_seq`

Examples

```
g <- cube_group(3)
s <- group_apply(g, group_identity(g), "R U R' U'")
sum(s != group_identity(g))

tg <- topspin_group(6, k = 3)
group_apply(tg, 1:6, c("1", "1", "3"))
```

group_compose *Compose a Move Sequence into One Permutation*

Description

Multiplies a word out into a single permutation of `1:n`. Applying that permutation once does what the whole word does, which is what makes repeating a word cheap.

Usage

```
group_compose(group, moves)
```

Arguments

`group`	A `perm_group`
`moves`	Move sequence: names, indices, or a space-separated string

Value

Integer vector of length `n`, the composed permutation

See Also

`group_apply`, `group_order`

Examples

```
g <- cube_group(3)
p <- group_compose(g, "R U R' U'")
identical(group_identity(g)[p], group_apply(g, group_identity(g), "R U R' U'"))
```

group_identity *Solved State of a Group*

Description

The identity, `1:n`, for whatever `n` the group works on.

Usage

```
group_identity(group)
```

Arguments

`group`	A `perm_group`

Value

Integer vector of length n

Examples

```
group_identity(topspin_group(6, 3))
length(group_identity(cube_group(3)))
```

`group_inverse_seq` *Invert a Move Sequence*

Description

The word that undoes the given one: reversed, each move replaced by its inverse. Errors if some move's inverse is not in the group's alphabet.

Usage

```
group_inverse_seq(group, moves, as_names = TRUE)
```

Arguments

group	A perm_group
moves	Move sequence: names, indices, or a space-separated string
as_names	Logical, return names rather than indices (default TRUE)

Value

The inverse sequence, as names or indices

See Also

group_apply

Examples

```
g <- cube_group(3)
group_inverse_seq(g, "R U R' U'")

# a word and its inverse cancel
s <- group_apply(g, group_identity(g), "R U U F'")
identical(group_apply(g, s, group_inverse_seq(g, "R U U F'")), group_identity(g))
```

group_move_index *Resolve Move Names to Indices Within a Group*

Description

Accepts moves however they are spelled — names, indices, or one space-separated string — and returns indices into the group's alphabet. Every function taking a `moves` argument runs it through here, which is why names and indices are interchangeable everywhere.

Usage

```
group_move_index(group, moves = NULL)
```

Arguments

`group`	A `perm_group`
`moves`	Move sequence: names, indices, or a space-separated string. `NULL` means the group's whole alphabet.

Details

Names are matched against the alphabet the group was built with, so a TopSpin group made with the default `c("1","2","3")` spelling does not answer to `"L"`: build it with `moves = c("L","R","X")` if that is the spelling you want to use.

Value

Integer vector of 1-based move indices

See Also

`group_moves`

Examples

```
g <- cube_group(3)
group_move_index(g, "R U R' U'")
group_move_index(g, c(4, 1, 5, 2))

# names resolve against the alphabet the group was built with
tg <- topspin_group(10, 4, moves = c("L", "R", "X"))
group_move_index(tg, c("L", "X"))

# the default spelling is "1","2","3", so those are its names
group_move_index(topspin_group(10, 4), c("1", "3"))
```

group_moves *Move Alphabet of a Group*

Description

Move Alphabet of a Group

Usage

```
group_moves(group)
```

Arguments

group	A perm_group

Value

Character vector of move names

Examples

```
group_moves(topspin_group(10, 4))
group_moves(cube_group(3))
```

group_order *Order of a Move Sequence in a Group*

Description

How many times a word must be repeated before the state returns to where it started. This is the order of the word as a group element, and does not depend on the starting state.

Usage

```
group_order(group, moves, max_reps = 100000L)
```

Arguments

group	A perm_group
moves	Move sequence: names, indices, or a space-separated string
max_reps	Integer, cap on repetitions before giving up (default 100000)

Value

Integer, the order, or NA if max_reps was reached

See Also

group_compose

Examples

```
g <- cube_group(3)
group_order(g, "R")
group_order(g, "R U")
group_order(g, "R U R' U'")
```

`group_parse_word` *Parse a Word Written as One String*

Description

Combination words have always been written unseparated — `"1231"` is four TopSpin operations — which reads only because each move is one character. A cube's moves are not: `"R'M'"` has to be split on its move names, not its characters. This does both, longest name first, so a group's own spelling always wins over a shorter prefix of it.

Usage

```
group_parse_word(group, word)
```

Arguments

`group`	A perm_group
`word`	Character string, or a vector of them

Value

For one string, an integer vector of move indices; for a vector, a list of them

See Also

group_move_index

Examples

```
group_parse_word(topspin_group(10, 4), "1231")
group_parse_word(cube_group(3), "RM'R'")
group_parse_word(cube_group(3), "R M' R'")
```

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 (`XLXLXRX` and `LXRXLXLX`, 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 *Human Algorithm Path Between Two Arbitrary States*

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

`start_state`	Integer vector, the starting permutation state
`target_state`	Integer vector, the target permutation state. Defaults to `1:n`, in which case this is exactly `human_algorithm`.
`k`	Integer, length of the reverse-prefix (flipper) operation
`simplify`	Logical, run `short_position()` 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` *Rank Candidate Moves by the Phase 1 Criterion*

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, `|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 *Invert a Path of Operations*

Description

Reverses a sequence of operations and replaces each by its inverse, giving the path that undoes the original. For TopSpin that means "1" (shift left) becomes "2" (shift right) and vice versa, while "3" (reverse) is its own inverse; for any other perm_group the group says which move undoes which.

Usage

```
invert_path(path, group = NULL)
```

Arguments

path	Character vector of operations
group	A perm_group. When NULL (default) the path is read as TopSpin's, as it always has been.

Value

Character vector of inverted operations in reverse order

See Also

group_inverse_seq

Examples

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

invert_path(c("R", "U", "U", "F'"), group = cube_group(3))
```

is_perm_group *Test Whether an Object is a Permutation Group*

Description

Test Whether an Object is a Permutation Group

Usage

```
is_perm_group(x)
```

Arguments

x	Any object

Value

Logical

Examples

```
is_perm_group(topspin_group(10, 4))
is_perm_group(1:10)
```

kociemba4_phases *The 4x4x4 reduction phases, one call at a time*

Description

The reduction that turns a 4x4x4 into a 3x3x3 runs in three phases, and these expose them individually. `cube_kociemba4_reduce` runs all three and reports only the result; these let a caller stop after any one of them, inspect what it produced, and decide what to do next – which is what `cube_solve4_cascade` does, and what every diagnostic in `inst/examples` was written against.

Usage

```
cube_kociemba4_phase12_cpp(state, upto_phase = 2L, max_depth1 = 10L,
  max_depth2 = 12L, node_budget = 5e7)

cube_kociemba4_phase3_cpp(state, max_depth3 = 14L, node_budget = 5e7,
  prune_depth_bonus = 0L, use_exact_centres = FALSE, progress_every = 0)

cube_at_phase_goal_cpp(state, phase)

cube_phase3_coord_cpp(state)
```

```
cube_to_pieces4_cpp(state)

cube_wing_geometry_cpp()

cube_kociemba4_tables_cpp()

cube_kociemba4_fill_phase3_cpp(depth, table_size = 0, breadth_first = TRUE,
  max_frontier = 4e+07)
```

Arguments

state	Integer vector of 96 stickers.
upto_phase	Run phase 1 only (1) or phases 1 and 2 (2).
max_depth1, max_depth2, max_depth3	Deepest search each phase may reach.
node_budget	Nodes a phase may visit before giving up. The outcome "exhausted" means this ran out, which is a statement about the budget and not about the cube: no path was found and none was ruled out.
prune_depth_bonus	Extra levels to fill the prune table beyond the default of half the search depth. Measured: each level costs about twenty-four times the last in fill time and does not move the bound off zero, because the entries collide before the depth matters.
use_exact_centres	Use the exact centre table (src/centre_table.h) alongside the hash table, taking the larger of the two bounds. The centre table is complete – 352,800 arrangements, every one with its own entry – so it cannot return the false zeroes the hash table returns when another state claims the slot.
phase	Which phase's goal to test, 1, 2 or 3.
progress_every	Print a line every this many nodes, naming the depth being searched and the nodes so far; 0 for silence.
depth	Depth to fill phase 3's prune table to.
table_size	Slots to grow the prune table to before filling; 0 keeps its current size.
breadth_first	Fill breadth-first (TRUE), visiting each state once, or depth-first (FALSE), which walks move words and revisits states – measured roughly a hundred times slower at depth 6.
max_frontier	Cap on the frontier the breadth-first fill holds in memory.

Value

cube_kociemba4_phase12_cpp returns the moves as a character vector.

cube_kociemba4_phase3_cpp returns a list: found, path, generators (the phase's own move words, which expand to path), states_perm and states_ori (the states the search passed through, one row per generator), outcome, best_bound, nodes and the prune counters. On failure the path and states describe the branch the table rated closest, which is where the effort went rather than progress towards a solution.

`cube_at_phase_goal_cpp` returns a single logical.

`cube_phase3_coord_cpp` returns phase 3's coordinate broken into parts, including `prune_bound` – which is only meaningful once the table has been built, so a fresh process reports it as 0 or 1 for everything.

`cube_to_pieces4_cpp` returns `perm` and `ori`: the 56-piece vector, with corners at 1:8, wings at 9:32 and centres at 33:56. Values carry the same offsets, so a wing's number runs 8..31 rather than 0..23.

`cube_wing_geometry_cpp` returns the wing geometry: `partner`, `dedge`, `primary`, `primary_in_dedge` and `speffz_to_slot`, all measured from the cube rather than tabulated.

`cube_kociemba4_tables_cpp` returns one entry per phase describing its prune table: `size`, `filled`, `built_depth`, `n_collisions`, `waste_ratio` and the per-depth counts.

See Also

`cube_kociemba4_reduce` for all three phases at once, `cube_solve4_cascade` for the bounded version that falls back.

`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 / 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: $1, h+1, 2, h+2, \ldots$ with $h = \lceil n/2 \rceil$.
4. **envelope** — taken alternately from the two ends towards the centre: $n, 1, n-1, 2, \ldots$.

5. **adjacent_swaps** — the full pairing of neighbours: swap(1,2), swap(3,4), ... A trailing odd element is left alone.
6. **broken_cycle** — the n-cycle $2, 3, \ldots, n, 1$ 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** — $1, n, 2, n-1, \ldots$, 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..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 *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)
}
```

---

openmp_threads *Number of OpenMP Threads Available*

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()
```

perm_group *Create a Permutation Group from a Table of Moves*

Description

Declares a group by listing its moves as explicit permutations. This is the general constructor: any puzzle whose moves permute a fixed set of positions can be expressed here, with no C++ involved.

Usage

```
perm_group(perms, n = NULL, name = "perm_group")
```

Arguments

perms	Named list of integer vectors, each a permutation of `1:n` read as `new[i] <- state[perm[i]]`. Names become the move names.
n	Integer, the state length. Defaults to the length of the first permutation.
name	Character, a label for printing (default `"perm_group"`)

Value

An object of class `perm_group`

See Also

topspin_group, cube_group, perm_group_intro

Examples

```
# the 3x3x3 cube, from the table the package already ships
g <- perm_group(cube_moves(3), n = 54, name = "cube3")
g

# a group of your own: two generators on five points
g2 <- perm_group(list(
  a = c(2L, 3L, 4L, 5L, 1L),
  b = c(2L, 1L, 3L, 4L, 5L)
))
group_moves(g2)
```

perm_group_intro *Permutation Groups*

Description

A perm_group is what the rest of the package means by "the puzzle being solved": a state length, an alphabet of named moves, and a way to apply one. Search functions take a group and work through it, so the same BFS, the same store and the same iterative solver run over TopSpin, over the 3x3x3 cube, or over any group you declare yourself.

Details

Two constructors cover the built-in puzzles — topspin_group and cube_group — and perm_group itself takes an arbitrary table of permutations, which is how a new puzzle enters without a line of C++.

What a group must supply

Moves are given as permutations of 1:n, each read as new[i] <- state[perm[i]]: entry i says where position i draws its new value from. This is the same convention cube_moves is written in, so that table can be handed over unchanged.

Inverses are worked out by composition, not by naming, so you need not say which move undoes which. A move whose inverse is absent from the alphabet is recorded as having none, and functions that need to walk a path backwards will say so rather than guess.

Moves versus alphabet

The group fixes the alphabet; a search may use any subset of it. Every search function still takes moves, now naming a subset of the group's moves rather than defining them. Leaving it out means the whole alphabet.

See Also

perm_group, topspin_group, cube_group

print.perm_group *Print a Permutation Group*

Description

Shows the group's name, the length of the states it works on, and its move alphabet.

Usage

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

Arguments

x	A perm_group
...	Ignored, present for consistency with print

Value

x, invisibly

See Also

perm_group, group_moves

Examples

```
print(topspin_group(12, k = 4))
cube_group(3)
```

---

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).

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 *Reverse First k Elements (Simple)*

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` *Length of the Sorted Run on the Ring*

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 *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).

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 *Shift State Left (Simple)*

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 *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).

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` *Shift State Right (Simple)*

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` *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 = NULL,
  depth = 5L,
  moves = NULL,
  group = NULL
)
```

Arguments

path	Move sequence to shorten, in the group's own spelling. For TopSpin that is "1"/"2"/"3" or "L"/"R"/"X" as before; for a cube, names such as "R" or "M'".
start_state	Integer vector, the starting permutation state
k	Integer, parameter for reverse_prefix operation. Ignored when group is given.
depth	Integer, BFS exploration depth (default 5)
moves	Allowed operations for the rewrite, naming a subset of the group's alphabet (default: all of it)
group	A perm_group. When NULL (default) the arguments describe TopSpin.

Details

Works over any perm_group: the shortener only needs to apply moves and compare states, so a cube solution shortens the same way a TopSpin path does. The result is spelled in the same move names as the input.

Value

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

See Also

perm_group

Examples

```
# a path with an obvious detour: L then R cancel
short_path_bfs(c("1", "2", "1", "3"), 1:8, k = 4)$path

# the same on a cube: R R' is a no-op
g <- cube_group(3)
short_path_bfs(c("R", "R'", "U"), group_identity(g), group = g)$path
```

---

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

Builds a sparse "highway" through the state space: at each level only a few candidates are kept, the highest-degree ones plus a random sample, so the search reaches deep without materialising the whole frontier.

Usage

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

Arguments

start_state	Integer vector — starting permutation
k	Integer — parameter for reverse_prefix operation. Ignored when group is given.
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)
moves	Allowed operations, naming a subset of the group's alphabet (default: all of it)
group	A perm_group. When NULL (default) the arguments describe TopSpin.

Details

Runs over any perm_group. Operations are reported under the group's own move names, so a cube highway is labelled with face turns.

Value

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

See Also

perm_group, reconstruct_bfs_path

Examples

```
set.seed(1)
head(sparse_bfs(1:10, k = 4, max_levels = 5))

# the same highway through the cube
g <- cube_group(3)
head(sparse_bfs(group_identity(g), group = g, max_levels = 3))
```

---

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` *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 = NULL,
  cycle_val,
  group = NULL,
  max_reps = 100000L
)
```

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. Ignored when the store carries a group, or when group is given.
cycle_val	Integer, cycle number to assign
group	A perm_group. Defaults to the store's own group, and to TopSpin if it has none.
max_reps	Integer, cap on repetitions of a word before its cycle is abandoned as non-closing (default 100000)

Details

Each combination is spun until the state returns to where it started, so a single short word yields its whole cycle. Words are written in the group's own spelling and may be unseparated ("1231", "RM'R'") or spaced.

Value

Number of states added (invisible)

See Also

perm_group, group_parse_word

---

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 *Drop All States From a Store*

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 *Clear All OPD Filters*

Description

Clear All OPD Filters

Usage

```
store_clear_opd(store)
```

Arguments

store	External pointer to StateStore

store_collect_ops *Collect Operations Leading to a State Within One Cycle*

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 *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_group *Group Attached to a State Store*

Description

Group Attached to a State Store

Usage

```
store_group(store)
```

Arguments

store	External pointer to StateStore

Value

The perm_group the store was created with, or NULL

Examples

```
store_group(create_state_store(54L, group = cube_group(3)))
```

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

topspin_group *Create the TopSpin Group*

Description

The group the package was built on: `L` rotates the ring one step left, `R` one step right, and `X` reverses the first `k` elements.

Usage

```
topspin_group(n, k = n, moves = c("1", "2", "3"))
```

Arguments

`n`	Integer, the number of positions on the ring
`k`	Integer, how many leading elements `X` reverses (default `n`, i.e. reverse the whole ring)
`moves`	Character vector naming the alphabet, in either spelling: `c("L","R","X")` or `c("1","2","3")` (default all three, as `"1","2","3"`, which is the spelling the package has always used)

Details

`k` is fixed here, at construction, and does not travel through the search functions afterwards — the group knows what its own `X` does.

Value

An object of class `perm_group`

See Also

`perm_group`, `cube_group`

Examples

```
g <- topspin_group(12, k = 4)
g
group_moves(g)

# a two-move alphabet: shifts only, no reversal
topspin_group(12, k = 4, moves = c("L", "R"))
```

`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 = NULL,
  group = NULL
)
```

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. Ignored when `group` is given.
`group`	A `perm_group`. When `NULL` (default) the path is read as TopSpin's.

Details

Simplification is TopSpin's: `short_position` reduces runs of shifts modulo the ring and cancels reverses, which are facts about L, R and X rather than about permutations in general. For any other group the path is verified but returned as it came in — checking is universal, this particular rewriting is not. `cycle_shortcut` and `short_path_bfs` shorten paths for every group.

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

g <- cube_group(3)
s <- group_apply(g, group_identity(g), "R U")
validate_and_simplify_path(c("R", "U"), group_identity(g), s, group = g)$valid
```

Index

analyze_top_combinations, 5
apply_operations, 6
apply_operations_batch_gpu, 7

bidirectional_bfs, 8
breakpoint_distance, 9
build_permutation_matrix, 10

calculate_angular_distance_z, 10
calculate_differences, 11
calculate_midpoint_z, 12
cayley_bfs_full, 12, *17*
cayley_distance *(*distance_methods*)*, 96
cayley_distance_methods *(*distance_methods*)*, 96
cayley_gpu_available, 14
cayley_gpu_free, 14
cayley_gpu_init, 15
cayley_gpu_status, 15
cayley_graph_diameter, *13*, 16
check_duplicates, 17
compose_permutation_matrix, 18
convert_digits, 18
convert_LRX_to_celestial, *13*, 19
convex_hull_3d, 20, *98*
create_state_store, 21, *141*
cube_adi, 22
cube_adi_astar, 23
cube_adi_load, 24, *26*
cube_adi_model, 25, *28*, *67*
cube_adi_save, *24*, 26
cube_adi_solve, *22–24*, 27
cube_adi_train, *22–24*, 27
cube_alg_table, 28, *46*
cube_apply_word, 29, *29*, *46*, *71*, *94*
cube_at_phase_goal_cpp *(*kociemba4_phases*)*, 124
cube_build_lslice, 30, *44, 45*, *66*
cube_central_moves, 31, *34*, *36*
cube_centre_counts, 32, *45*, *51*
cube_centre_cycles, *31*, 33
cube_centre_map_family, 34, *37*, *80*
cube_centre_positions, 35
cube_centre_shots, *31, 32*, *34*, 36, *44, 45*, *61*
cube_centre_structure, *32–36*, 37, *42*, *61*, *79, 80*
cube_colour_state *(*cube_colours*)*, 38
cube_colours, 38
cube_edge_algs, 39, *66*, *93*
cube_edge_counts, 40, *40*, *66*
cube_edge_map, 41, *42, 43*
cube_edge_map_family, *41*, 42, *43*, *66*
cube_edge_structure, *40–42*, 43
cube_empty_u_slice, 44
cube_expand_alg, 45
cube_expand_move, 46
cube_expand_word *(*cube_expand_move*)*, 46
cube_free_moves *(*cube_lslice_cells*)*, 60
cube_group, *47*, 48, *48*, *62, 63*, *65, 66*, *74*, *81*, *83–86*, *89*, *108*, *110*, *130, 131*, *151*
cube_identity, *35*, 49, *50*, *63*
cube_is_colour_solved, *38*, *49*, 50, *63*, *78*, *81*
cube_is_reduced, *40*, 51, *59*, *72*, *88*
cube_kociemba, 52, *54, 55*, *59*, *88*
cube_kociemba4, 55, *59*
cube_kociemba4_fill_phase3_cpp *(*kociemba4_phases*)*, 124
cube_kociemba4_phase12_cpp *(*kociemba4_phases*)*, 124
cube_kociemba4_phase3_cpp *(*kociemba4_phases*)*, 124
cube_kociemba4_reduce, *55, 56*, 57, *57*, *59*, *126*
cube_kociemba4_report, *57, 58*, 59
cube_kociemba4_tables_cpp *(*kociemba4_phases*)*, 124
cube_kociemba_init, 54
cube_kociemba_report, *52, 53*, 55, *59*

cube_layer_move, 59
cube_lslice_cells, *30, 31*, 60
cube_move_names, *47, 48*, 61, *80*
cube_moves, *34*, *41, 42*, *47*, *60*, 62, *62, 63*, *73, 74*, *93*, *131*
cube_moves_santa *(*cube_santa_moves*)*, 75
cube_nnn, *48*, *50*, *62*, 63, *63*, *74*
cube_orbits, *34, 35*, *37*, 64, *65*, *68*, *70*
cube_orbits_family, 64, *64*
cube_pair_edges, 65
cube_phase3_coord_cpp *(*kociemba4_phases*)*, 124
cube_piece_layout, *26*, 66
cube_pieces, *38*, *42, 43*, *51*, *64–66*, 67, *67*
cube_pieces_home, 68, *70*
cube_predicates, 69, *71*
cube_progress, *64, 65*, *68*, 69
cube_read_state, *29*, *69*, 70
cube_reduce_best, 71
cube_reduce_cpp, *71*, 72, *72*, *79*
cube_santa, 73, *74–78*
cube_santa_group, 74
cube_santa_move_names, 75
cube_santa_moves, *74*, 75, *75*, *77*
cube_santa_path, 76
cube_santa_path_out *(*cube_santa_path*)*, 76
cube_santa_perm, *74*, *76*, 77, *78*
cube_santa_state, *38*, *74*, *77*, 78
cube_santa_state_out, *74*
cube_santa_state_out *(*cube_santa_state*)*, 78
cube_slice_cells_cpp, 79
cube_slice_map, *35*, *37*, *41*, *79*, 80
cube_solve4, *51*, *56–59*, *72, 73*, 86
cube_solve4_cascade, 89, *124*, *126*
cube_solve_centres, *32, 33*, 81, *89*
cube_solve_cfop, *29*, *52, 53*, *69*, 82, *83–86*, *89*
cube_solve_lbl, *53*, 83, *83*, *85, 86*
cube_solve_m2, *81*, 84
cube_solve_old_pochmann, *84*, 85, *85*
cube_to_pieces4_cpp *(*kociemba4_phases*)*, 124
cube_wide_move *(*cube_expand_move*)*, 46
cube_wide_turn, *39, 40*, 92
cube_wide_word *(*cube_expand_move*)*, 46
cube_wing_geometry_cpp *(*kociemba4_phases*)*, 124
cube_word_order, *29*, 93
cycle_shortcut, 94, *129*, *152*

distance_methods, 96, *99, 100*

enclosing_hull_3d, 97

find_best_match_cube4_model, *97*, 98
find_best_match_human, *97*, 99, *99*
find_best_random_combinations, *95*, 100, *106*
find_closest_to_coords, 102
find_combination_in_states, 103
find_path_bfs, 103
find_path_iterative, *94*, *96, 97*, 105, *121*

generate_state, *32*, *51, 52*, *57*, *81–86*, 107, *109, 110*
generate_unique_states_df, 108
generate_walk, 109
get_reachable_states, 111
get_reachable_states_light, 112
group_apply, 113, *114, 115*
group_compose, *113*, 114, *118*
group_identity, 114
group_inverse_seq, *113*, 115, *123*
group_move_index, 116, *118*
group_moves, *116*, 117, *132*
group_order, *114*, 117
group_parse_word, 118, *143*

human_algorithm, 119, *120–123*, *134*
human_algorithm_to, *96*, 120
human_phase1_navigate, 121
human_phase1_rank, *97*, *100*, *121*, 122, *134*

invert_path, *109, 110*, 123
is_perm_group, 124

kociemba4_phases, 124

landmark_states, 126

manhattan_distance, 128
manhattan_distance_matrix_gpu, 128

openmp_threads, 129

perm_group, *8*, *13*, *16*, *21*, *26*, *48*, *95*, *101*, *104, 105*, *107*, *109*, *112–118*, *123*,

130, *131, 132*, *139–141*, *143*, *148*, *151, 152*
perm_group_intro, *130*, 131
print, *132*
print.perm_group, 131

reconstruct_bfs_path, 132, *141*
reverse_prefix, 133
reverse_prefix_simple, 133
run_length, *122, 123*, 134

save_bridge_states, 135
select_unique, 135
shift_left, 136
shift_left_simple, 137
shift_right, 137
shift_right_simple, 138
short_path_bfs, *87*, *95, 96*, 138, *152*
short_position, 139, *152*
sparse_bfs, *12, 13*, 140
state_store_indices_for_cycle *(*state_store_query*)*, 141
state_store_perm_length *(*state_store_query*)*, 141
state_store_query, 141
state_store_size *(*state_store_query*)*, 141
state_store_unique_count *(*state_store_query*)*, 141
store_add_from_df, 142
store_analyze_combos, 142
store_analyze_combos_gpu, 143
store_clear, 144, *145*
store_clear_opd, 144
store_collect_ops, 145
store_combos_for_state, 145
store_filter_middle, 146
store_find_best_match, 146
store_find_intersections, 147
store_get_meta, 147
store_get_state, 148
store_group, 148
store_lookup, 149
store_reconstruct_path, 149
store_set_opd, 150
store_to_dataframe, 150

topspin_group, *130, 131*, 151

validate_and_simplify_path, 152