```
open Printf


(* ----- GLOBALS & CONSTS ----- *)

let debug = false  (* Whether to print debug data or not *)

type hashtable_data = {
    path: string;
    size: int;
    depth: int;
    mutable hashtable: (int, int) Hashtbl.t;
    mutable loaded: bool;
}

let corners_hashtable = {
    path = "Heuristiques Korf/corners_hashtable";
    size = 88_179_840;
    depth = max_int;
    hashtable = Hashtbl.create 0;
    loaded = false;
}
let edges_fst_hashtable = {
    path = "Heuristiques Korf/edges_fst_half_hashtable";
    size = 42_577_920;
    depth = max_int;
    hashtable = Hashtbl.create 0;
    loaded = false;
}
let edges_snd_hashtable = {
    path = "Heuristiques Korf/edges_snd_half_hashtable";
    size = 42_577_920;
    depth = max_int;
    hashtable = Hashtbl.create 0;
    loaded = false;
}

let korf_hashtable = {  (* Gathers "corners_hashtable" (hash > 0), and
        "edges_fst_half_hashtable" (hash < 0) *)
    path = "Heuristiques Korf/korf_hashtable";
    size = 130_757_760;
    depth = max_int;
    hashtable = Hashtbl.create 0;
    loaded = false;
}

let g1_thistlethwaite_hashtable = {
    path = "Heuristiques Thistlethwaite/g1";
    size = 2_048;
    depth = max_int;
    hashtable = Hashtbl.create 0;
    loaded = false;
}
let g2_thistlethwaite_hashtable = {
    path = "Heuristiques Thistlethwaite/g2";
    size = 1_082_565;
    depth = max_int;
    hashtable = Hashtbl.create 0;
    loaded = false;
}
```

```
let g3_thistlethwaite_hashtable = {
    path = "Heuristiques Thistlethwaite/g3";
    size = 2_822_400;
    depth = max_int;
    hashtable = Hashtbl.create 0;
    loaded = false;
}
let g4_thistlethwaite_hashtable = {
    path = "Heuristiques Thistlethwaite/g4";
    size = 663_552;
    depth = max_int;
    hashtable = Hashtbl.create 0;
    loaded = false;
}

let g2_kociemba_hashtable = {
    path = "Heuristiques Kociemba/g2";
    size = 1;
    depth = 7;
    hashtable = Hashtbl.create 0;
    loaded = false;
}
let g4_kociemba_hashtable = {
    path = "Heuristiques Kociemba/g4";
    size = 1;
    depth = 9;
    hashtable = Hashtbl.create 0;
    loaded = false;
}


type move = {
    face: int;  (* 0 - 5 *)
    nb_dir: int;  (* 0 - 3 *)
}
let moves_list =
    let list = ref [] in
    for face = 0 to 5 do
        for nb_dir = 1 to 3 do
            list := {face = face; nb_dir = nb_dir} :: !list
        done
    done;
    !list

let number_of_corners = 8
type corner = {
    corner_id: int;  (* 0 - 7;  8 for empty corner *)
    orientation: int;  (* 0 - 2 *)
}

let number_of_edges = 12
type edge = {
    edge_id: int;  (* 0 - 11;  12 for empty edge *)
    orientation: int;  (* 0 - 1 *)
}

type cube = {
    corners: corner array;
    edges: edge array;
}


(* ----- UTILS ----- *)
```

```
let rec pow_int x n =
    if n = 0 then 1
    else if n mod 2 = 0 then pow_int (x * x) (n / 2)
    else x * pow_int x (n - 1)

let string_of_char c =
    String.make 1 c

let char_of_face f =
    match f with
    | 0 -> 'W'
    | 1 -> 'R'
    | 2 -> 'Y'
    | 3 -> 'O'
    | 4 -> 'B'
    | 5 -> 'G'
    | _ -> failwith "char_of_face"

let face_of_char c =
    match c with
    | 'W' -> 0
    | 'R' -> 1
    | 'Y' -> 2
    | 'O' -> 3
    | 'B' -> 4
    | 'G' -> 5
    | _ -> failwith "face_of_char"


(* ----- CUBE ----- *)

let default_cube () =
    {
        corners = Array.init number_of_corners (fun x -> {corner_id = x; orientation = 0});
        edges = Array.init number_of_edges (fun x -> {edge_id = x; orientation = 0});
    }

let copy_cube cube =
    {corners = Array.copy cube.corners; edges = Array.copy cube.edges}

let empty_corner_id =  (* `number_of_corners` (8) is the id of empty corners *)
    number_of_corners

let empty_corner =
    {corner_id = empty_corner_id; orientation = 0}

let empty_edge_id =  (* `number_of_edges` (12) is the id of empty edges *)
    number_of_edges

let empty_edge =
    {edge_id = empty_edge_id; orientation = 0}

let empty_cube () =
    {corners = Array.make number_of_corners empty_corner;
        edges = Array.make number_of_edges empty_edge}

let corners_permutation face =  (* Corners permutation for each move:
        corners.(i).id <- array.(i) *)
    match face with
    | 0 -> [|3; 0; 1; 2; 4; 5; 6; 7|]
    | 1 -> [|1; 5; 2; 3; 0; 4; 6; 7|]
    | 2 -> [|0; 1; 2; 3; 5; 6; 7; 4|]
    | 3 -> [|0; 1; 3; 7; 4; 5; 2; 6|]
    | 4 -> [|4; 1; 2; 0; 7; 5; 6; 3|]
    | 5 -> [|0; 2; 6; 3; 4; 1; 5; 7|]
    | _ -> failwith "corners_permutation"

let corners_orientation face =  (* Corner orientation for each move (before permutation):
        corners.(i).orientation += array.(i) (mod 3) *)
    match face with
    | 0 -> [|2; 1; 2; 1; 0; 0; 0; 0|]
    | 1 -> [|0; 0; 0; 0; 0; 0; 0; 0|]
    | 2 -> [|0; 0; 0; 0; 1; 2; 1; 2|]
    | 3 -> [|0; 0; 0; 0; 0; 0; 0; 0|]
    | 4 -> [|1; 0; 0; 2; 2; 0; 0; 1|]
    | 5 -> [|0; 2; 1; 0; 0; 1; 2; 0|]
    | _ -> failwith "corners_orientation"

let edges_permutation face =  (* Edges permutation for each move:
        edges.(i).id <- array.(i) *)
    match face with
    | 0 -> [|3; 0; 1; 2; 4; 5; 6; 7; 8; 9; 10; 11|]
    | 1 -> [|0; 9; 2; 3; 4; 8; 6; 7; 1; 5; 10; 11|]
    | 2 -> [|0; 1; 2; 3; 5; 6; 7; 4; 8; 9; 10; 11|]
    | 3 -> [|0; 1; 2; 11; 4; 5; 6; 10; 8; 9; 3; 7|]
    | 4 -> [|8; 1; 2; 3; 11; 5; 6; 7; 4; 9; 10; 0|]
    | 5 -> [|0; 1; 10; 3; 4; 5; 9; 7; 8; 2; 6; 11|]
    | _ -> failwith "edges_permutation"

let edges_orientation face =  (* Edges orientation for each move (before permutation):
        edges.(i).orientation += array.(i) (mod 2) *)
    match face with
    | 0 -> [|1; 1; 1; 1; 0; 0; 0; 0; 0; 0; 0; 0|]
    | 1 -> [|0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0|]
    | 2 -> [|0; 0; 0; 0; 1; 1; 1; 1; 0; 0; 0; 0|]
    | 3 -> [|0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0|]
    | 4 -> [|0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0|]
    | 5 -> [|0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0|]
    | _ -> failwith "edges_orientation"

let rec make_move cube move =
    let cube = if move.nb_dir > 1 then make_move cube {face = move.face; nb_dir = move.nb_dir - 1}
        else cube
    in
    let new_cube = empty_cube () in

    let corners_permutation = corners_permutation move.face in
    let corners_orientation = corners_orientation move.face in
    for i = 0 to number_of_corners - 1 do
        let new_corner = cube.corners.(corners_permutation.(i)) in
        let corner_id = new_corner.corner_id in
        let orientation = (new_corner.orientation + corners_orientation.(i)) mod 3 in
        new_cube.corners.(i) <- {corner_id = corner_id; orientation = orientation};
    done;

    let edges_permutation = edges_permutation move.face in
    let edges_orientation = edges_orientation move.face in
    for i = 0 to number_of_edges - 1 do
        let new_edge = cube.edges.(edges_permutation.(i)) in
        let edge_id = new_edge.edge_id in
        let orientation = (new_edge.orientation + edges_orientation.(i)) mod 2 in
        new_cube.edges.(i) <- {edge_id = edge_id; orientation = orientation};
    done;

    new_cube
```

```
let hash_corners cube =  (* Total size: 48 bits *)
    let h = ref 0 in
    for corner_location = 0 to number_of_corners - 1 do
        let corner = cube.corners.(corner_location) in
        h := !h lsl 4;
        h := !h + corner.corner_id;  (* 4 bits *)  (* Not 3 due to the empty corner (id = 8) *)
        h := !h lsl 2;
        h := !h + corner.orientation;  (* 2 bits *)
    done;
    !h

let hash_edges cube =  (* Total size: 60 bits *)
    let h = ref 0 in
    for edge_location = 0 to number_of_edges - 1 do
        let edge = cube.edges.(edge_location) in
        h := !h lsl 4;
        h := !h + edge.edge_id;  (* 4 bits *)
        h := !h lsl 1;
        h := !h + edge.orientation;  (* 1 bits *)
    done;
    !h

let hash_cube cube =
    (hash_corners cube, hash_edges cube)

let unhash_corners corners_hashed =
    let corners_hashed = ref corners_hashed in
    let corners = Array.make number_of_corners empty_corner in
    for corner_location = number_of_corners - 1 downto 0 do
        let corner_orientation = !corners_hashed mod 4 in  (* 2^2 *)
        corners_hashed := !corners_hashed lsr 2;
        let corner_id = !corners_hashed mod 16 in  (* 2^4 *)
        corners_hashed := !corners_hashed lsr 4;
        corners.(corner_location) <- {corner_id = corner_id; orientation = corner_orientation};
    done;
    corners

let unhash_edges edges_hashed =
    let edges_hashed = ref edges_hashed in
    let edges = Array.make number_of_edges empty_edge in
    for edge_location = number_of_edges - 1 downto 0 do
        let edge_orientation = !edges_hashed mod 2 in  (* 2^1 *)
        edges_hashed := !edges_hashed lsr 1;
        let edge_id = !edges_hashed mod 16 in  (* 2^4 *)
        edges_hashed := !edges_hashed lsr 4;
        edges.(edge_location) <- {edge_id = edge_id; orientation = edge_orientation};
    done;
    edges

let unhash_cube cube_hashed =
    let corners_hashed, edges_hashed = cube_hashed in
    {corners = unhash_corners corners_hashed; edges = unhash_edges edges_hashed}

let print_moves_list path =
    let path = Array.of_list path in
    for i = 0 to Array.length path - 1 do
        let {face = face; nb_dir = nb_dir} = path.(i) in
        printf "%d) %c, %d\n" (i + 1) (char_of_face face) nb_dir;
    done;
    printf "Solution lenght: %d\n" (Array.length path)

let random_cube nb_moves =
    let cube = ref (default_cube ()) in
    let last_face = ref None in

    for i = 1 to nb_moves do
        let face = ref (Random.int 6) in
        while Some !face = !last_face do
            face := Random.int 6
        done;
        last_face := Some !face;
        let nb_dir = ref ((Random.int 3) + 1) in  (* From 1 to 3 *)
        cube := make_move !cube {face = !face; nb_dir = !nb_dir};
    done;
    !cube

let full_random_cube () =
    let n = Random.int 10000 in
    random_cube n


(* ----- PRINT / READ ----- *)

let corner_colors_of_id id =  (* Takes a corner id, then enumerates its colors in the "natural order"
        for orientation = 0  (Natural order: orientation = 0; = 1; = 2) *)
    match id with
    | 0 -> (1, 4, 0)
    | 1 -> (1, 0, 5)
    | 2 -> (3, 5, 0)
    | 3 -> (3, 0, 4)
    | 4 -> (1, 2, 4)
    | 5 -> (1, 5, 2)
    | 6 -> (3, 2, 5)
    | 7 -> (3, 4, 2)
    | _ -> failwith "corner_colors_of_id"
let arr_id_of_corner_id id =  (* Location in the face of each color for a given corner,
        again in the "natural order" *)
    match id with
    | 0 -> (0, 8, 2)
    | 1 -> (6, 8, 2)
    | 2 -> (8, 0, 6)
    | 3 -> (2, 0, 6)
    | 4 -> (2, 0, 2)
    | 5 -> (8, 8, 6)
    | 6 -> (6, 8, 6)
    | 7 -> (0, 0, 2)
    | _ -> failwith "arr_id_of_corner_id"
let edge_colors_of_id id =  (* Same, for edges *)
    match id with
    | 0 -> (4, 0)
    | 1 -> (0, 1)
    | 2 -> (5, 0)
    | 3 -> (0, 3)
    | 4 -> (4, 2)
    | 5 -> (2, 1)
    | 6 -> (5, 2)
    | 7 -> (2, 3)
    | 8 -> (4, 1)
    | 9 -> (5, 1)
    | 10 -> (5, 3)
    | 11 -> (4, 3)
    | _ -> failwith "edge_colors_of_id"
let arr_id_of_edge_id id =  (* Same, for edges *)
    match id with
    | 0 -> (7, 1)
    | 1 -> (5, 3)
```

```
        | 2 -> (1, 7)
        | 3 -> (3, 5)
        | 4 -> (1, 1)
        | 5 -> (3, 5)
        | 6 -> (7, 7)
        | 7 -> (5, 3)
        | 8 -> (5, 1)
        | 9 -> (5, 7)
        | 10 -> (3, 7)
        | 11 -> (3, 1)
        | _ -> failwith "arr_id_of_edge_id"

let swap3 (a, b, c) n =  (* Adds an orientation of n on the corner *)
    if n = 0 then a, b, c
    else if n = 1 then b, c, a
    else if n = 2 then c, a, b
    else failwith "swap3"
let swap2 (a, b) n =  (* Same, for edges *)
    if n = 0 then a, b
    else if n = 1 then b, a
    else failwith "swap2"

let equal3 t1 t2 =  (* To check if two corners are the same (same colors in a different order) *)
    if t1 = swap3 t2 0 then true, 0
    else if t1 = swap3 t2 1 then true, 1
    else if t1 = swap3 t2 2 then true, 2
    else false, -1
let equal2 t1 t2 =  (* Same, for edges *)
    if t1 = swap2 t2 0 then true, 0
    else if t1 = swap2 t2 1 then true, 1
    else false, -1

let id_of_corner_colors colors =  (* Inverse function of `corner_colors_of_id` *)
    let r = ref (0, 0) in
    for id = 0 to number_of_corners - 1 do
        let corner_colors_of_id = corner_colors_of_id id in
        let eq, swap = equal3 colors corner_colors_of_id in
        if eq then
            r := id, swap;
    done;
    !r
let id_of_edge_colors colors =  (* Same, for edges *)
    let r = ref (0, 0) in
    for id = 0 to number_of_edges - 1 do
        let edge_colors_of_id = edge_colors_of_id id in
        let eq, swap = equal2 colors edge_colors_of_id in
        if eq then
            r := id, swap;
    done;
    !r

let array_of_cube cube =
    let arr = Array.make_matrix 6 9 (-1) in
    for face = 0 to 5 do
        arr.(face).(4) <- face;
    done;
    for corner_loc = 0 to number_of_corners - 1 do
        let cubie = cube.corners.(corner_loc) in

        let real_a, real_b, real_c = swap3 (corner_colors_of_id cubie.corner_id) cubie.orientation in
        let loc_a, loc_b, loc_c = corner_colors_of_id corner_loc in
        let arr_id_a, arr_id_b, arr_id_c = arr_id_of_corner_id corner_loc in
        assert (arr.(loc_a).(arr_id_a) = -1);  (* We should not overwrite any previous data *)
        assert (arr.(loc_b).(arr_id_b) = -1);
        assert (arr.(loc_c).(arr_id_c) = -1);

        arr.(loc_a).(arr_id_a) <- real_a;
        arr.(loc_b).(arr_id_b) <- real_b;
        arr.(loc_c).(arr_id_c) <- real_c;
    done;
    for edge_loc = 0 to number_of_edges - 1 do
        let cubie = cube.edges.(edge_loc) in

        let real_a, real_b = swap2 (edge_colors_of_id cubie.edge_id) cubie.orientation in
        let loc_a, loc_b = edge_colors_of_id edge_loc in
        let arr_id_a, arr_id_b = arr_id_of_edge_id edge_loc in

        assert (arr.(loc_a).(arr_id_a) = -1);  (* We should not overwrite any previous data *)
        assert (arr.(loc_b).(arr_id_b) = -1);

        arr.(loc_a).(arr_id_a) <- real_a;
        arr.(loc_b).(arr_id_b) <- real_b;
    done;
    arr

let print_cube_array arr =
    let str_of_int i =
        if i = -1 then "_"
        else string_of_char (char_of_face i)
    in
    for i = 0 to 2 do
        Printf.printf "      ";  (* 6 spaces *)
        for j = 0 to 2 do
            Printf.printf "%s " (str_of_int arr.(4).(i * 3 + j))
        done;
        Printf.printf "\n"
    done;
    for i = 0 to 2 do
        let f = 3 in
        for j = 0 to 2 do
            Printf.printf "%s " (str_of_int arr.(f).(i * 3 + j))
        done;
        for f = 0 to 2 do
            for j = 0 to 2 do
                Printf.printf "%s " (str_of_int arr.(f).(i * 3 + j))
            done
        done;
        Printf.printf "\n"
    done;
    for i = 0 to 2 do
        Printf.printf "      ";
        for j = 0 to 2 do
            Printf.printf "%s " (str_of_int arr.(5).(i * 3 + j))
        done;
        Printf.printf "\n"
    done;
    print_newline ()

let print_cube cube =
    print_cube_array (array_of_cube cube)

let print_cube_array_to_read arr out_channel =
    for face = 0 to 5 do
        for i = 0 to 8 do
            fprintf out_channel "%c " (char_of_face arr.(face).(i));
        done;
        fprintf out_channel "\n";
    done

let print_cube_to_read cube out_channel =
    print_cube_array_to_read (array_of_cube cube) out_channel

let read_cube_array in_channel =
    let arr = Array.make_matrix 6 9 (-1) in

    let channel = Scanf.Scanning.from_channel in_channel in
    for face = 0 to 5 do
        for i = 0 to 8 do
            arr.(face).(i) <- Scanf.bscanf channel "%c " (fun c -> face_of_char c);
        done;
    done;
    arr

let read_cube_array_from_formated_channel channel =
    let arr = Array.make_matrix 6 9 (-1) in

    for face = 0 to 5 do
        for i = 0 to 8 do
            arr.(face).(i) <- Scanf.bscanf channel "%c " (fun c -> face_of_char c);
        done;
    done;
    arr

let cube_of_array arr =
    let cube = empty_cube () in

    let corners = cube.corners in
    for corner_loc = 0 to number_of_corners - 1 do
        let loc_a, loc_b, loc_c = corner_colors_of_id corner_loc in
        let arr_id_a, arr_id_b, arr_id_c = arr_id_of_corner_id corner_loc in

        let real_a = arr.(loc_a).(arr_id_a) in
        let real_b = arr.(loc_b).(arr_id_b) in
        let real_c = arr.(loc_c).(arr_id_c) in

        let corner_id, swap = id_of_corner_colors (real_a, real_b, real_c) in

        assert (corners.(corner_loc).corner_id = empty_corner_id);  (* We should not overwrite
            any previous data *)

        corners.(corner_loc) <- {corner_id = corner_id; orientation = swap}
    done;

    let edges = cube.edges in
    for edge_loc = 0 to number_of_edges - 1 do
        let loc_a, loc_b = edge_colors_of_id edge_loc in
        let arr_id_a, arr_id_b = arr_id_of_edge_id edge_loc in

        let real_a = arr.(loc_a).(arr_id_a) in
        let real_b = arr.(loc_b).(arr_id_b) in

        let edge_id, swap = id_of_edge_colors (real_a, real_b) in

        assert (edges.(edge_loc).edge_id = empty_edge_id);  (* We should not overwrite
            any previous data *)

        edges.(edge_loc) <- {edge_id = edge_id; orientation = swap}
    done;
    cube

let read_cube in_channel =
    cube_of_array (read_cube_array in_channel)

let read_cube_from_formated_channel channel =
    cube_of_array (read_cube_array_from_formated_channel channel)


(* ----- BOTH ALGORITHMS ----- *)

(* Explanation of number encoding:
    input/output_binary_int use only 32-bits ints
    OCaml integers are encoded on 63 bits (1 for the sign, plus 62 bits)
    So, we can split them in 32 and 31 bits. Thus, the sign will be stored by the 32-bits number
*)

let save_hashtable hashtable file_name compact =
    if compact then (
        let file_name = file_name ^ ".bin" in

        if Sys.file_exists file_name then
            Sys.remove file_name;
        let out_channel_bin = open_out_bin file_name in
        Hashtbl.iter (fun cube_hashed dist ->
            output_binary_int out_channel_bin (cube_hashed lsr 31);  (* 32 bits: 32 - 63 *)
            let mask = lnot (1 lsl 31) in  (* 1 everywhere, 0 at the 32nd bit *)
            output_binary_int out_channel_bin ((cube_hashed mod (1 lsl 31)) land mask);
                (* 31 bits: 1 - 31 ; 32nd bit set to 0 to avoid negative number when reading *)
            output_byte out_channel_bin dist
        ) hashtable;
        close_out out_channel_bin;

        if debug then
            printf "Heuristic \"%s\" saved, %d values\n" file_name (Hashtbl.length hashtable);
    ) else (
        let file_name = file_name ^ ".txt" in

        if Sys.file_exists file_name then
            Sys.remove file_name;
        let out_channel = open_out file_name in
        Hashtbl.iter (fun cube_hashed dist ->
            fprintf out_channel "%s:%d\n" (string_of_int cube_hashed) dist
        ) hashtable;
        close_out out_channel;

        if debug then
            printf "Heuristic \"%s\" saved, %d values\n" file_name (Hashtbl.length hashtable);
    )

let compute_hashtable moves_list default_cube_list keep_funct hash unhash hashtable_data =
    let t0 = Sys.time () in

    let hashtable = Hashtbl.create 1 in
    let q = Queue.create () in
    List.iter (fun c ->
        Queue.add (hash c, 0) q;
        Hashtbl.add hashtable (hash c) 0;
    ) default_cube_list;

    let treatment cube dist =
        if dist + 1 <= hashtable_data.depth then
```

```
            List.iter (fun move ->
                let new_cube = keep_funct (make_move cube move) in
                let hashed_new_cube = hash new_cube in
                if not (Hashtbl.mem hashtable hashed_new_cube) then (
                    Queue.add (hashed_new_cube, dist + 1) q;
                    Hashtbl.add hashtable (hashed_new_cube) (dist + 1);
                )
            ) moves_list
    in

    while not (Queue.is_empty q) do
        let (hashed_cube, dist) = Queue.pop q in
        let cube = unhash hashed_cube in
        treatment cube dist
    done;

    if debug then
        printf "Heuristic \"%s\" computed, %d values (%fs)\n" hashtable_data.path
            (Hashtbl.length hashtable) (Sys.time () -. t0);
    save_hashtable hashtable hashtable_data.path true;
    save_hashtable hashtable hashtable_data.path false

let load_hashtable hashtable_data compact =
    let t0 = Sys.time () in
    let hashtable = Hashtbl.create hashtable_data.size in
    if compact then (
        let file_name = hashtable_data.path ^ ".bin" in
        let in_channel_bin = open_in_bin file_name in
        try
            while true do
                let h1 = input_binary_int in_channel_bin in
                let h2 = input_binary_int in_channel_bin in
                let h = (h1 lsl 31) + h2 in
                let d = input_byte in_channel_bin in
                Hashtbl.add hashtable h d
            done;
            failwith ("Error: load_hashtable: " ^ file_name)
        with
        | End_of_file ->
            close_in in_channel_bin;
            if debug then
                printf "Heuristic \"%s\" loaded, %d values (%fs)\n" file_name
                    (Hashtbl.length hashtable) (Sys.time () -. t0);
    ) else (
        let file_name = hashtable_data.path ^ ".txt" in
        let in_channel = open_in file_name in
        try
            let channel = Scanf.Scanning.from_channel in_channel in
            while true do
                let h, d = Scanf.bscanf channel "%d:%d\n" (fun h d -> h, d) in
                Hashtbl.add hashtable h d
            done;
            failwith ("Error: load_hashtable: " ^ file_name)
        with
        | End_of_file ->
            close_in in_channel;
            if debug then
                printf "Heuristic \"%s\" loaded, %d values (%fs)\n" file_name
                    (Hashtbl.length hashtable) (Sys.time () -. t0);
    );
    hashtable_data.hashtable <- hashtable;
    hashtable_data.loaded <- true
```

```
let unload_hashtable hashtable_data =
    hashtable_data.hashtable <- Hashtbl.create 0;
    hashtable_data.loaded <- false

let ida_star cube moves_list keep_funct heuristic =
    let t0 = Sys.time () in

    let minimum = ref 0 in
    let minimum_reached = ref max_int in
    let exception Found of move list in

    let is_mirror a b =
        if a = 0 then b = 2
        else if a = 1 then b = 3
        else if a = 2 then b = 0
        else if a = 3 then b = 1
        else if a = 4 then b = 5
        else if a = 5 then b = 4
        else false
    in
    let rec dfs cube dist last_move path =
        let heuristic_value = heuristic (unhash_cube cube) in
        let dist_min = dist + heuristic_value in
        if heuristic_value = 0 then
            raise (Found path)
        else if dist_min > !minimum then
            minimum_reached := min !minimum_reached dist_min
        else (  (* dist_min = !minimum *)
            List.iter (fun move ->
                if (last_move.face <> move.face) && ((not (is_mirror move.face last_move.face))
                        || move.face < last_move.face) then (  (* for exemple, UD = DU *)
                    let cube = hash_cube (keep_funct (make_move (unhash_cube cube) move)) in
                    dfs cube (dist + 1) move (move :: path)
                )
            ) moves_list
        )
    in

    try
        while !minimum <= 20 do  (* Every cube can be solved in 20 moves or less *)
            minimum_reached := max_int;
            dfs (hash_cube (keep_funct cube)) 0 {face = -1; nb_dir = -1} [];
            if debug then printf "Not under: %d ; Time: %fs\n" !minimum_reached (Sys.time () -. t0);
            minimum := !minimum_reached
        done;
        failwith "Error: Cube impossible to solve"
    with
    | Found path -> path

let a_star cube moves_list heuristic =
    let t0 = Sys.time () in

    let exception Found in

    let heap = Heap.create () in
    Heap.insert heap (hash_cube cube, 0);

    let seen = Hashtbl.create 1 in
    Hashtbl.add seen (hash_cube cube) 0;

    let last_move = Hashtbl.create 1 in

    let reached_dist = ref 0 in
    try
        while Heap.get_min heap <> None do
            let hashed_cube, dist = Option.get (Heap.extract_min heap) in
            if debug then (
                if dist > !reached_dist then (
                    reached_dist := dist;
                    printf "Not under: %d ; Time: %fs\n" !reached_dist (Sys.time () -. t0);
                );
            );
            let cube = unhash_cube hashed_cube in
            let cube_dist = Hashtbl.find seen hashed_cube in
            let heuristic_value = heuristic cube in
            if heuristic_value = 0 then
                raise Found;
            List.iter (fun move ->
                let new_cube = make_move cube move in
                let hashed_new_cube = hash_cube new_cube in
                let current_min_dist = cube_dist + 1 in
                match Hashtbl.find_opt seen hashed_new_cube with
                | Some d when d <= current_min_dist -> ()
                | _ ->
                    Hashtbl.replace seen hashed_new_cube current_min_dist;
                    Hashtbl.replace last_move hashed_new_cube move;
                    Heap.insert_or_decrease heap
                        (hashed_new_cube, current_min_dist + heuristic new_cube);
            ) moves_list
        done;
        failwith "Error: Cube impossible to solve"
    with
    | Found ->
        let cube = ref (default_cube ()) in
        let path = ref [] in
        let move = ref (Hashtbl.find_opt last_move (hash_cube !cube)) in
        while !move <> None do
            path := Option.get !move :: !path;
            let {face = face; nb_dir = nb_dir} = Option.get !move in
            cube := make_move !cube {face = face; nb_dir = 4 - nb_dir};  (* Inverted move *)
            move := Hashtbl.find_opt last_move (hash_cube !cube);
        done;
        List.rev !path

let bfs cube =
    let queue = Queue.create () in
    Queue.add (hash_cube cube) queue;
    let last_move = Hashtbl.create 1 in
    let exception Found in
    try
        while not (Queue.is_empty queue) do
            let cube_hashed = Queue.pop queue in
            if cube_hashed = hash_cube (default_cube ()) then
                raise Found;
            let cube = unhash_cube cube_hashed in
            List.iter (fun move ->
                let new_cube = make_move cube move in
                let hashed_new_cube = hash_cube new_cube in
                if Hashtbl.find_opt last_move hashed_new_cube = None then
                    Hashtbl.add last_move hashed_new_cube move;
                Queue.add (hash_cube new_cube) queue;
            ) moves_list
        done;
        failwith "Error: Cube impossible to solve"
    with
    | Found ->
```

```
        Hashtbl.remove last_move (hash_cube cube);
        let cube = ref (default_cube ()) in
        let path = ref [] in
        let move = ref (Hashtbl.find_opt last_move (hash_cube !cube)) in
        while !move <> None do
            path := Option.get !move :: !path;
            let {face = face; nb_dir = nb_dir} = Option.get !move in
            cube := make_move !cube {face = face; nb_dir = 4 - nb_dir};  (* Inverted move *)
            move := Hashtbl.find_opt last_move (hash_cube !cube);
        done;
        List.rev !path


(* ----- KORF'S ALGORITHM ----- *)

let fst_half_edges_array = [|0; 1; 2; 3; 8; 9|]
let snd_half_edges_array = [|6; 7; 4; 5; 10; 11|]

let keep_corners cube =
    let cube = copy_cube cube in
    {corners = cube.corners; edges = Array.make number_of_edges empty_edge}

let keep_fst_half_edges cube =
    let cube = copy_cube cube in
    let edges = cube.edges in
    for i = 0 to number_of_edges - 1 do
        if not (Array.mem edges.(i).edge_id fst_half_edges_array) then
            edges.(i) <- empty_edge;
    done;
    {corners = Array.make number_of_corners empty_corner; edges = edges}

let keep_snd_half_edges cube =
    let cube = copy_cube cube in
    let edges = cube.edges in
    for i = 0 to number_of_edges - 1 do
        if not (Array.mem edges.(i).edge_id snd_half_edges_array) then
            edges.(i) <- empty_edge;
    done;
    {corners = Array.make number_of_corners empty_corner; edges = edges}

let fst_equ_of_snd cube =
    let flip_cube_edges cube =
        let new_cube = copy_cube cube in
        for i = 0 to Array.length fst_half_edges_array - 1 do
            new_cube.edges.(fst_half_edges_array.(i)) <- cube.edges.(snd_half_edges_array.(i));
            new_cube.edges.(snd_half_edges_array.(i)) <- cube.edges.(fst_half_edges_array.(i));
        done;

        for edge_loc = 0 to number_of_edges - 1 do
            if Array.mem new_cube.edges.(edge_loc).edge_id fst_half_edges_array then (
                for i = 0 to Array.length fst_half_edges_array - 1 do
                    if new_cube.edges.(edge_loc).edge_id = fst_half_edges_array.(i) then
                        new_cube.edges.(edge_loc) <- {edge_id = snd_half_edges_array.(i);
                            orientation = new_cube.edges.(edge_loc).orientation}
                done;
            ) else (
                for i = 0 to Array.length snd_half_edges_array - 1 do
                    if new_cube.edges.(edge_loc).edge_id = snd_half_edges_array.(i) then
                        new_cube.edges.(edge_loc) <- {edge_id = fst_half_edges_array.(i);
                            orientation = new_cube.edges.(edge_loc).orientation}
                done;
            )
        done;
        new_cube
    in
    let flipped_cube = flip_cube_edges cube in
    keep_fst_half_edges {corners = Array.make number_of_corners empty_corner;
        edges = flipped_cube.edges}

let compute_korf_hashtable keep_funct hash unhash hashtable_data =
    compute_hashtable moves_list [keep_funct (default_cube ())] keep_funct hash unhash hashtable_data

let load_korf_hashtable hashtable_data compact =
    load_hashtable hashtable_data compact

let recompute_korf_hashtables () =
    compute_korf_hashtable keep_fst_half_edges hash_edges
        (fun cube_hash -> unhash_cube (0, cube_hash)) edges_fst_hashtable;
    compute_korf_hashtable keep_snd_half_edges hash_edges
        (fun cube_hash -> unhash_cube (0, cube_hash)) edges_snd_hashtable;
    compute_korf_hashtable keep_corners hash_corners
        (fun cube_hash -> unhash_cube (cube_hash, 0)) corners_hashtable;

    load_korf_hashtable corners_hashtable true;
    load_korf_hashtable edges_fst_hashtable true;
    let h = Hashtbl.create korf_hashtable.size in
    Hashtbl.iter (fun k v ->
        Hashtbl.add h k v
    ) corners_hashtable.hashtable;
    Hashtbl.iter (fun k v ->
        Hashtbl.add h (- k) v
    ) edges_fst_hashtable.hashtable;
    save_hashtable h korf_hashtable.path true;
    save_hashtable h korf_hashtable.path false

let unload_korf_hashtables () =
    unload_hashtable corners_hashtable;
    unload_hashtable edges_fst_hashtable;
    unload_hashtable edges_snd_hashtable;
    unload_hashtable korf_hashtable

let korf_heuristic hashtable cube =
    let max3 a b c =
        max (max a b) c
    in

    (* let cor = Hashtbl.find corners_hashtable (hash_corners (keep_corners cube)) in
    let fst = Hashtbl.find edges_fst_hashtable (hash_edges (keep_fst_half_edges cube)) in
    let snd = Hashtbl.find edges_snd_hashtable (hash_edges (keep_snd_half_edges cube)) in *)

    (* let cor = Hashtbl.find corners_hashtable (hash_corners (keep_corners cube)) in
    let fst = Hashtbl.find edges_fst_hashtable (hash_edges (keep_fst_half_edges cube)) in
    let snd = Hashtbl.find edges_fst_hashtable
        (hash_edges (fst_equ_of_snd (keep_snd_half_edges cube))) in *)

    let cor = Hashtbl.find hashtable (hash_corners (keep_corners cube)) in
    let fst = Hashtbl.find hashtable (- (hash_edges (keep_fst_half_edges cube))) in
    let snd = Hashtbl.find hashtable (- (hash_edges (fst_equ_of_snd (keep_snd_half_edges cube)))) in

    max3 cor fst snd

let korf cube =
    if debug then (
        printf "Korf (IDA*):\n";
        print_cube cube;
    );
    let compact = true in

    (* recompute_korf_hashtables (); *)

    (* let cor = load_korf_hashtable corners_hashtable_path corners_hashtable_size compact in *)
    (* let fst = load_korf_hashtable edges_fst_hashtable_path edges_fst_hashtable_size compact in *)
    (* let snd = load_korf_hashtable edges_snd_hashtable_path edges_snd_hashtable_size compact in *)

    if not korf_hashtable.loaded then
        load_korf_hashtable korf_hashtable compact;

    let heuristic = korf_heuristic korf_hashtable.hashtable in
    let keep_funct c = c in

    ida_star cube moves_list keep_funct heuristic


(* ----- A* SOLVER ----- *)

let a_star_solver cube =
    if debug then (
        printf "A*:\n";
        print_cube cube;
    );

    let compact = true in

    if not korf_hashtable.loaded then
        load_korf_hashtable korf_hashtable compact;

    let heuristic = korf_heuristic korf_hashtable.hashtable in

    a_star cube moves_list heuristic


(* ----- THISTLETHWAITE'S ALGORITHM ----- *)

let fst_corner_orbit = [|0; 2; 5; 7|]
let snd_corner_orbit = [|1; 3; 4; 6|]
let fst_slice_edges = [|0; 2; 4; 6|]
let snd_slice_edges = [|1; 3; 5; 7|]
let trd_slice_edges = [|8; 9; 10; 11|]

(* G1: Good edges orientation *)  (* Faces 0-2 can only be moved two by two *)
(* G2: Edges of the first slice (1-3 : L-R) in this slice AND Good corners orientation *)
    (* Faces 4-5 can only be moved two by two *)
(* G3: Reachable from default cube with G3 moves only: Each edge in its slice AND Each corner in its
    orbit (NECESSARY, NOT SUFFICIENT, PROPERTY) *)  (* Faces 1-3 can only be moved two by two *)
(* G4: Solved cube *)

let true_is_gn n =
    let true_is_g1 cube =
        let result = ref true in
        for i = 0 to number_of_edges - 1 do
            if cube.edges.(i).orientation <> 0 then
                result := false;
        done;
        !result
    in

    let true_is_g2 cube =
        let result = ref true in
        for i = 0 to number_of_corners - 1 do
```

```
            if cube.corners.(i).orientation <> 0 then
                result := false;
        done;
        for i = 0 to Array.length fst_slice_edges - 1 do
            let edge_loc = fst_slice_edges.(i) in
            if not (Array.mem cube.edges.(edge_loc).edge_id fst_slice_edges) then
                result := false;
        done;
        !result
    in

    let true_is_g3 cube =  (* WARNING: THIS IS A NECESSARY, NOT SUFFICIENT, PROPERTY *)
        let result = ref true in
        for i = 0 to Array.length fst_corner_orbit - 1 do
            let corner_loc = fst_corner_orbit.(i) in
            if not (Array.mem cube.corners.(corner_loc).corner_id fst_corner_orbit) then
                result := false;
        done;
        for i = 0 to Array.length snd_corner_orbit - 1 do
            let corner_loc = snd_corner_orbit.(i) in
            if not (Array.mem cube.corners.(corner_loc).corner_id snd_corner_orbit) then
                result := false;
        done;
        for i = 0 to Array.length snd_slice_edges - 1 do
            let edge_loc = snd_slice_edges.(i) in
            if not (Array.mem cube.edges.(edge_loc).edge_id snd_slice_edges) then
                result := false;
        done;
        for i = 0 to Array.length trd_slice_edges - 1 do
            let edge_loc = trd_slice_edges.(i) in
            if not (Array.mem cube.edges.(edge_loc).edge_id trd_slice_edges) then
                result := false;
        done;
        !result
    in

    let true_is_g4 cube =
        hash_cube cube = hash_cube (default_cube ())
    in

    match n with
    | 1 -> true_is_g1
    | 2 -> true_is_g2
    | 3 -> true_is_g3
    | 4 -> true_is_g4
    | _ -> failwith "true_is_gn"


let hash_gn_thistlethwaite n =
    let hash_g1 cube =  (* Total size: 12 bits *)
        let h = ref 0 in
        for edge_location = 0 to number_of_edges - 1 do
            h := !h lsl 1;
            h := !h + cube.edges.(edge_location).orientation;
        done;
        !h
    in

    let hash_g2 cube =  (* Total size: 28 bits *)
        let h = ref 0 in
        for corner_location = 0 to number_of_corners - 1 do
            h := !h lsl 2;
            h := !h + cube.corners.(corner_location).orientation;
        done;
        done;
        for edge_location = 0 to number_of_edges - 1 do
            h := !h lsl 1;
            if Array.mem cube.edges.(edge_location).edge_id fst_slice_edges then
                h := !h + 1;
        done;
        !h
    in

    let hash_g3 cube =  (* Total size: 40 bits *)
        let h = ref 0 in
        for corner_location = 0 to number_of_corners - 1 do
            h := !h lsl 4;
            h := !h + cube.corners.(corner_location).corner_id;
        done;
        for edge_location = 0 to number_of_edges - 1 do
            if not (Array.mem edge_location fst_slice_edges) then (
                h := !h lsl 1;
                if Array.mem cube.edges.(edge_location).edge_id snd_slice_edges then
                    h := !h + 1
                else
                    assert (Array.mem cube.edges.(edge_location).edge_id trd_slice_edges);
            );
        done;
        !h
    in

    let hash_g4 cube =  (* Total size: 56 bits *)
        let h = ref 0 in
        for corner_location = 0 to number_of_corners - 1 do
            h := !h lsl 4;
            h := !h + cube.corners.(corner_location).corner_id;
        done;
        let aux slice =
            for i = 0 to Array.length slice - 1 do
                h := !h lsl 2;
                let edge_location = slice.(i) in
                let edge_id = cube.edges.(edge_location).edge_id in
                for j = 0 to Array.length slice - 1 do
                    if edge_id = slice.(j) then
                        h := !h + j
                done;
            done;
        in
        aux fst_slice_edges;
        aux snd_slice_edges;
        aux trd_slice_edges;
        !h
    in

    match n with
    | 1 -> hash_g1
    | 2 -> hash_g2
    | 3 -> hash_g3
    | 4 -> hash_g4
    | _ -> failwith "hash_gn_thistlethwaite"


let unhash_gn_thistlethwaite n =
    let unhash_g1 h =
        let h = ref h in
        let edges = Array.make number_of_edges empty_edge in
        for edge_location = number_of_edges - 1 downto 0 do
            let orientation = !h mod 2 in
            edges.(edge_location) <- {edge_id = empty_edge_id; orientation = orientation};
            h := !h lsr 1;
        done;
        {corners = Array.make number_of_corners empty_corner; edges = edges}
    in

    let unhash_g2 h =
        let h = ref h in
        let corners = Array.make number_of_corners empty_corner in
        let edges = Array.make number_of_edges empty_edge in
        for edge_location = number_of_edges - 1 downto 0 do
            let is_slice = !h mod 2 in
            if is_slice = 1 then
                edges.(edge_location) <- {edge_id = fst_slice_edges.(0); orientation = 0}
            else
                edges.(edge_location) <- {edge_id = empty_edge_id; orientation = 0};
            h := !h lsr 1
        done;
        for corner_location = number_of_corners - 1 downto 0 do
            let orientation = !h mod 4 in
            corners.(corner_location) <- {corner_id = empty_corner_id; orientation = orientation};
            h := !h lsr 2;
        done;
        {corners = corners; edges = edges}
    in

    let unhash_g3 h =
        let h = ref h in
        let corners = Array.make number_of_corners empty_corner in
        let edges = Array.make number_of_edges empty_edge in
        for edge_location = number_of_edges - 1 downto 0 do
            if not (Array.mem edge_location fst_slice_edges) then (
                let edge_snd_slice = !h mod 2 in
                if edge_snd_slice = 1 then
                    edges.(edge_location) <- {edge_id = snd_slice_edges.(0); orientation = 0}
                else
                    edges.(edge_location) <- {edge_id = trd_slice_edges.(0); orientation = 0};
                h := !h lsr 1
            ) else (
                edges.(edge_location) <- {edge_id = fst_slice_edges.(0); orientation = 0};
            );
        done;
        for corner_location = number_of_corners - 1 downto 0 do
            let corner_id = !h mod 16 in
            corners.(corner_location) <- {corner_id = corner_id; orientation = 0};
            h := !h lsr 4;
        done;
        {corners = corners; edges = edges}
    in

    let unhash_g4 h =
        let h = ref h in
        let corners = Array.make number_of_corners empty_corner in
        let edges = Array.make number_of_edges empty_edge in
        let aux slice =
            for i = Array.length slice - 1 downto 0 do
                let j = !h mod 4 in
                edges.(slice.(i)) <- {edge_id = slice.(j); orientation = 0};
                h := !h lsr 2
            done;
        in
        aux trd_slice_edges;
        aux snd_slice_edges;
        aux fst_slice_edges;
        for corner_location = number_of_corners - 1 downto 0 do
            let corner_id = !h mod 16 in
            corners.(corner_location) <- {corner_id = corner_id; orientation = 0};
            h := !h lsr 4
        done;
        {corners = corners; edges = edges}
    in

    match n with
    | 1 -> unhash_g1
    | 2 -> unhash_g2
    | 3 -> unhash_g3
    | 4 -> unhash_g4
    | _ -> failwith "hash_gn_thistlethwaite"


let keep_gn_thistlethwaite n cube =
    unhash_gn_thistlethwaite n (hash_gn_thistlethwaite n cube)


let moves_list_to_gn_thistlethwaite n =
    let moves_list_to_g1 =
        moves_list
    in

    let moves_list_to_g2 =
        List.filter (fun move ->
            if (move.face = 0 || move.face = 2) && (move.nb_dir mod 2 = 1) then
                false
            else true
        ) moves_list_to_g1
    in

    let moves_list_to_g3 =
        List.filter (fun move ->
            if (move.face = 4 || move.face = 5) && (move.nb_dir mod 2 = 1) then
                false
            else true
        ) moves_list_to_g2
    in

    let moves_list_to_g4 =
        List.filter (fun move ->
            if (move.nb_dir mod 2 = 1) then
                false
            else true
        ) moves_list_to_g3
    in

    match n with
    | 1 -> moves_list_to_g1
    | 2 -> moves_list_to_g2
    | 3 -> moves_list_to_g3
    | 4 -> moves_list_to_g4
    | _ -> failwith "moves_list_to_gn"


let compute_thistlethwaite_hashtable moves_list default_cube_list hash unhash hashtable_data =
    let keep_funct c = unhash (hash c) in
    compute_hashtable moves_list default_cube_list keep_funct hash unhash hashtable_data
```

```
let load_thistlethwaite_hashtable hashtable_data compact =
    load_hashtable hashtable_data compact


let g3_default_cube_list_thistlethwaite () =
    let hash_g3 = hash_gn_thistlethwaite 3 in
    let unhash_g3 = unhash_gn_thistlethwaite 3 in
    let moves_list_to_g4 = moves_list_to_gn_thistlethwaite 4 in
    let cube = ref (default_cube ()) in
    let q = Queue.create () in
    Queue.add (hash_g3 !cube) q;
    let seen = Hashtbl.create 1 in

    while Queue.length q > 0 do
        let cube_hashed = Queue.pop q in
        if not (Hashtbl.mem seen cube_hashed) then (
            Hashtbl.add seen cube_hashed true;
            cube := unhash_g3 cube_hashed;
            List.iter (fun move ->
                let c = make_move !cube move in
                Queue.add (hash_g3 c) q
            ) moves_list_to_g4;
        )
    done;
    let l = ref [] in
    Hashtbl.iter (fun c _ -> l := (unhash_g3 c) :: !l) seen;
    !l


let target_list_to_gn_thistlethwaite n =
    let default_cube = default_cube () in
    match n with
    | 1 ->
        let keep_g1 = keep_gn_thistlethwaite n in
        [keep_g1 default_cube]
    | 2 ->
        let keep_g2 = keep_gn_thistlethwaite n in
        [keep_g2 default_cube]
    | 3 ->
        g3_default_cube_list_thistlethwaite ()
    | 4 ->
        let keep_g4 = keep_gn_thistlethwaite n in
        [keep_g4 default_cube]
    | _ -> failwith "target_to_gn_thistlethwaite"


let gn_hashtable_thistlethwaite n =
    match n with
    | 1 -> g1_thistlethwaite_hashtable
    | 2 -> g2_thistlethwaite_hashtable
    | 3 -> g3_thistlethwaite_hashtable
    | 4 -> g4_thistlethwaite_hashtable
    | _ -> failwith "gn_hashtable_thistlethwaite"


let recompute_thistlethwaite_hashtables () =
    for n = 1 to 4 do
        let moves_list_to_gn = moves_list_to_gn_thistlethwaite n in
        let target_list_to_gn = target_list_to_gn_thistlethwaite n in
        let hash_gn = hash_gn_thistlethwaite n in
        let unhash_gn = unhash_gn_thistlethwaite n in
        let gn_hashtable = gn_hashtable_thistlethwaite n in
        compute_thistlethwaite_hashtable moves_list_to_gn target_list_to_gn hash_gn unhash_gn
```

```
            gn_hashtable
    done


let load_thistlethwaite_hashtables compact =
    for n = 1 to 4 do
        let gn_hashtable = gn_hashtable_thistlethwaite n in
        if not gn_hashtable.loaded then
            load_thistlethwaite_hashtable gn_hashtable compact;
    done


let unload_thistlethwaite_hashtables () =
    for n = 1 to 4 do
        let gn_hashtable = gn_hashtable_thistlethwaite n in
        if gn_hashtable.loaded then
            unload_hashtable gn_hashtable;
    done


let thistlethwaite_ida_star cube =
    if debug then (
        printf "Thistlethwaite (IDA*):\n";
        print_cube cube;
    );

    let cube = ref cube in
    let path = ref [] in

    let compact = true in
    (* recompute_thistlethwaite_hashtables (); *)
    load_thistlethwaite_hashtables compact;

    let h_gn n =
        match n with
        | 1 -> g1_thistlethwaite_hashtable.hashtable
        | 2 -> g2_thistlethwaite_hashtable.hashtable
        | 3 -> g3_thistlethwaite_hashtable.hashtable
        | 4 -> g4_thistlethwaite_hashtable.hashtable
        | _ -> failwith "h_gn"
    in

    if debug then print_newline ();

    let process_to_gn n =
        if debug then printf "G: %d -> %d\n" (n - 1) n;

        let hash_gn = hash_gn_thistlethwaite n in
        let keep_gn = keep_gn_thistlethwaite n in
        let h_gn = h_gn n in
        let moves_list_to_gn = moves_list_to_gn_thistlethwaite n in

        let solve_gn cube =
            let heuristic_gn cube =
                try
                    Hashtbl.find h_gn (hash_gn cube)
                with
                | Not_found -> printf "%d\n" (hash_gn cube); max_int
            in
            ida_star cube moves_list_to_gn keep_gn heuristic_gn
        in

        let path_gn = solve_gn !cube in
        path := path_gn @ !path;
        List.iter (fun move -> cube := make_move !cube move) (List.rev path_gn);
        if debug then print_cube !cube;
        assert (true_is_gn n !cube);
    in

    for i = 1 to 4 do
        process_to_gn i
    done;

    if debug then (
        printf "Solved!\n";
        print_cube !cube;
        print_moves_list (List.rev !path);
        print_newline ();
        print_newline ();
    );

    List.rev !path


let thistlethwaite_bfs cube =
    if debug then (
        printf "Thistlethwaite (BFS):\n";
        print_cube cube;
    );

    let cube = ref cube in
    let path = ref [] in

    let process_to_gn n =
        if debug then printf "G: %d -> %d\n" (n - 1) n;
        let hash_gn = hash_gn_thistlethwaite n in
        let unhash_gn = unhash_gn_thistlethwaite n in
        let moves_list_to_gn = moves_list_to_gn_thistlethwaite n in
        let target_list_to_gn = target_list_to_gn_thistlethwaite n in
        let target_hashed_to_gn = List.map (fun c -> hash_gn c) target_list_to_gn in

        let solve_gn cube =
            let cube = ref cube in
            let path_gn = ref [] in
            let q = Queue.create () in
            let seen = Hashtbl.create 1 in
            Queue.add (hash_gn !cube, []) q;

            while not (List.mem (hash_gn !cube) target_hashed_to_gn) do
                assert (Queue.length q > 0);
                let cube_hashed, p = Queue.pop q in
                if not (Hashtbl.mem seen cube_hashed) then (
                    Hashtbl.add seen cube_hashed true;
                    cube := unhash_gn cube_hashed;
                    path_gn := p;
                    List.iter (fun move ->
                        let c = make_move !cube move in
                        let hashed_c = hash_gn c in
                        if not (Hashtbl.mem seen hashed_c) then
                            Queue.add (hashed_c, move :: p) q
                    ) moves_list_to_gn;
                )
            done;
            !path_gn
        in
        let path_gn = solve_gn !cube in
        path := path_gn @ !path;
        List.iter (fun move -> cube := make_move !cube move) (List.rev path_gn);
        if debug then print_cube !cube;
        assert (true_is_gn n !cube);
    in

    for n = 1 to 4 do
        process_to_gn n
    done;

    if debug then (
        printf "Solved!\n";
        print_cube !cube;
        print_moves_list (List.rev !path);
        print_newline ();
    );

    List.rev !path


(* ----- KOCIEMBA'S ALGORITHM ----- *)

(* G2: Edges of the first slice (1-3 : L-R) in this slice AND Good corners orientation AND
    Good edges orientation *)  (* Faces 0-2 and 4-5 can only be moved two by two *)
(* G4: Solved cube *)

let hash_gn_kociemba n =
    let hash_g2 cube =  (* Total size: 36 bits *)
        let h = ref 0 in
        for edge_location = 0 to number_of_edges - 1 do
            h := !h lsl 1;
            h := !h + cube.edges.(edge_location).orientation;
        done;
        for corner_location = 0 to number_of_corners - 1 do
            h := !h lsl 2;
            h := !h + cube.corners.(corner_location).orientation;
        done;
        for edge_location = 0 to number_of_edges - 1 do
            h := !h lsl 1;
            if Array.mem cube.edges.(edge_location).edge_id fst_slice_edges then
                h := !h + 1;
        done;
        !h
    in

    let hash_g4 cube =  (* Total size: 62 bits (61.6) *)
        let h = ref 0 in
        for corner_location = 0 to number_of_corners - 1 do
            h := !h lsl 3;
            h := !h + cube.corners.(corner_location).corner_id;
        done;
        for i = 0 to Array.length fst_slice_edges - 1 do
            h := !h lsl 2;
            for j = 0 to Array.length fst_slice_edges - 1 do
                if cube.edges.(fst_slice_edges.(i)).edge_id = fst_slice_edges.(j) then
                    h := !h + j;
            done;
        done;
        for edge_location = 0 to number_of_edges - 1 do
            if not (Array.mem edge_location fst_slice_edges) then (
                h := !h * 13;  (* Not !h lsl 4 as it would makes a 64 bits hash, and int are Int63 *)
                h := !h + cube.edges.(edge_location).edge_id;
            )
        done;
        !h
    in

    match n with
    | 2 -> hash_g2
    | 4 -> hash_g4
    | _ -> failwith "hash_gn_kociemba"


let unhash_gn_kociemba n =
    let unhash_g2 h =
        let h = ref h in
        let corners = Array.make number_of_corners empty_corner in
        let edges = Array.make number_of_edges empty_edge in
        for edge_location = number_of_edges - 1 downto 0 do
            let is_slice = !h mod 2 in
            if is_slice = 1 then
                edges.(edge_location) <- {edge_id = fst_slice_edges.(0); orientation = 0}
            else
                edges.(edge_location) <- {edge_id = empty_edge_id; orientation = 0};
            h := !h lsr 1
        done;
        for corner_location = number_of_corners - 1 downto 0 do
            let orientation = !h mod 4 in
            corners.(corner_location) <- {corner_id = empty_corner_id; orientation = orientation};
            h := !h lsr 2;
        done;
        for edge_location = number_of_edges - 1 downto 0 do
            let orientation = !h mod 2 in
            edges.(edge_location) <- {edge_id = edges.(edge_location).edge_id;
                orientation = orientation};
            h := !h lsr 1;
        done;
        {corners = corners; edges = edges}
    in

    let unhash_g4 h =
        let h = ref h in
        let corners = Array.make number_of_corners empty_corner in
        let edges = Array.make number_of_edges empty_edge in
        for edge_location = number_of_edges - 1 downto 0 do
            if not (Array.mem edge_location fst_slice_edges) then (
                let edge_id = !h mod 13 in
                edges.(edge_location) <- {edge_id = edge_id; orientation = 0};
                h := !h / 13;
            );
        done;
        for i = Array.length fst_slice_edges - 1 downto 0 do
            let j = !h mod 4 in
            let edge_id = fst_slice_edges.(j) in
            edges.(fst_slice_edges.(i)) <- {edge_id = edge_id; orientation = 0};
            h := !h lsr 2;
        done;
        for corner_location = number_of_corners - 1 downto 0 do
            let corner_id = !h mod 8 in
            corners.(corner_location) <- {corner_id = corner_id; orientation = 0};
            h := !h lsr 3;
        done;
        {corners = corners; edges = edges}
    in

    match n with
    | 2 -> unhash_g2
    | 4 -> unhash_g4
    | _ -> failwith "unhash_gn_kociemba"


let keep_gn_kociemba n cube =
    unhash_gn_kociemba n (hash_gn_kociemba n cube)


let moves_list_to_gn_kociemba n =
    let moves_list_to_g2 =
        moves_list
    in

    let moves_list_to_g4 =
        List.filter (fun move ->
            if (move.face = 0 || move.face = 2 || move.face = 4 || move.face = 5)
                    && (move.nb_dir mod 2 = 1) then
                false
            else true
        ) moves_list_to_g2
    in

    match n with
    | 2 -> moves_list_to_g2
    | 4 -> moves_list_to_g4
    | _ -> failwith "moves_list_to_gn_kociemba"


let compute_kociemba_hashtable moves_list default_cube_list hash unhash hashtable_data =
    let keep_funct c = unhash (hash c) in
    compute_hashtable moves_list default_cube_list keep_funct hash unhash hashtable_data


let gn_hashtable_kociemba n =
    match n with
    | 2 -> g2_kociemba_hashtable
    | 4 -> g4_kociemba_hashtable
    | _ -> failwith "gn_hashtable_kociemba"


let recompute_kociemba_hashtables () =
    for n = 1 to 2 do
        let n = 2 * n in
        let moves_list_to_gn = moves_list_to_gn_kociemba n in
        let hash_gn = hash_gn_kociemba n in
        let unhash_gn = unhash_gn_kociemba n in
        let gn_hashtable_kociemba = gn_hashtable_kociemba n in
        compute_kociemba_hashtable moves_list_to_gn [unhash_gn (hash_gn (default_cube ()))]
            hash_gn unhash_gn gn_hashtable_kociemba
    done


let load_kociemba_hashtable hashtable_data compact =
    load_hashtable hashtable_data compact


let load_kociemba_hashtables compact =
    for n = 1 to 2 do
        let n = 2 * n in
        let gn_hashtable = gn_hashtable_kociemba n in
        if not gn_hashtable.loaded then
            load_kociemba_hashtable gn_hashtable compact;
    done
```

```
let unload_kociemba_hashtables () =
    for n = 1 to 2 do
        let n = 2 * n in
        let gn_hashtable = gn_hashtable_kociemba n in
        if not gn_hashtable.loaded then
            unload_hashtable gn_hashtable;
    done


let kociemba_ida_star cube =
    if debug then (
        printf "Kociemba (IDA*):\n";
        print_cube cube;
    );

    let cube = ref cube in
    let path = ref [] in

    let compact = true in
    (* recompute_kociemba_hashtables (); *)
    load_kociemba_hashtables compact;

    let h_gn n =
        match n with
        | 2 -> g2_kociemba_hashtable.hashtable
        | 4 -> g4_kociemba_hashtable.hashtable
        | _ -> failwith "h_gn"
    in

    if debug then print_newline ();

    let process_to_gn n =
        if debug then printf "G: %d -> %d\n" (n - 2) n;
        let hash_gn = hash_gn_kociemba n in
        let keep_gn = keep_gn_kociemba n in
        let h_gn = h_gn n in
        let moves_list_to_gn = moves_list_to_gn_kociemba n in
        let max_len = (gn_hashtable_kociemba n).depth in

        let solve_gn cube =
            let heuristic_gn cube =
                match Hashtbl.find_opt h_gn (hash_gn cube) with
                | Some n -> n
                | None -> max_len + 1
            in
            assert (heuristic_gn (keep_gn (default_cube ())) = 0);
            ida_star cube moves_list_to_gn keep_gn heuristic_gn
        in

        let path_gn = solve_gn !cube in
        path := path_gn @ !path;
        List.iter (fun move -> cube := make_move !cube move) (List.rev path_gn);
        if debug then print_cube !cube;
        assert (true_is_gn n !cube);
    in

    for i = 1 to 2 do
        process_to_gn (2 * i)
    done;

    if debug then (
        printf "Solved!\n";
        print_cube !cube;
        print_moves_list (List.rev !path);
        print_newline ();
        print_newline ();
    );

    List.rev !path


(* ----- RESULTS ANALYSIS ----- *)

let get_time_data f cubes =
    f (default_cube ());  (* Inits hashtables *)
    let n = Array.length cubes in
    let times = Array.make n (-1.) in
    for i = 0 to n - 1 do
        let t0 = Sys.time () in
        f cubes.(i);
        let t1 = Sys.time () in
        times.(i) <- t1 -. t0;
    done;
    times

let get_path_len_data f cubes =
    f (default_cube ());  (* Inits hashtables *)
    let n = Array.length cubes in
    let len = Array.make n (-1) in
    for i = 0 to n - 1 do
        len.(i) <- List.length (f cubes.(i));
    done;
    len

let full_random_cubes n =
    Array.init n (fun i -> full_random_cube ())

let random_cubes n nb_moves =
    Array.init n (fun i -> random_cube nb_moves)

let save_cubes cubes file_name =
    if Sys.file_exists file_name then
        Sys.remove file_name;
    let file = open_out file_name in
    for i = 0 to Array.length cubes - 1 do
        print_cube_to_read cubes.(i) file;
    done;
    close_out file

let read_cubes nb_moves n =
    let file_name = ("Cubes/" ^ (string_of_int nb_moves) ^ ".txt") in
    let file = open_in file_name in
    let cubes = Array.make n (default_cube ()) in
    let channel = Scanf.Scanning.from_channel file in
    for i = 0 to n - 1 do
        cubes.(i) <- read_cube_from_formated_channel channel;
    done;
    close_in file;
    cubes

let generate_cubes nb_cubes_per_depth =  (* e.g: [|0; 1; 2; 2|] for 0 of 0 move, 1 of 1 move,
        2 of 2 moves and 2 of 3 moves *)
    let max_depth = Array.length nb_cubes_per_depth - 1 in
    for depth = 1 to max_depth do
```

```
        let cubes = random_cubes nb_cubes_per_depth.(depth) depth in
        save_cubes cubes ("Cubes/" ^ (string_of_int depth) ^ ".txt");
    done

let get_and_print_time_data f nb_cubes_per_depth algo_name =
    printf "%s\n" algo_name;
    let max_depth = Array.length nb_cubes_per_depth - 1 in
    printf "%d\n" max_depth;
    for depth = 1 to max_depth do
        let cubes = read_cubes depth nb_cubes_per_depth.(depth) in
        let data = get_time_data f cubes in
        let len = Array.length data in
        printf "%d\n" len;
        for i = 0 to len - 1 do
            printf "%f\n" data.(i);
        done;
    done

let get_and_print_path_len_data f nb_cubes_per_depth algo_name =
    printf "%s\n" algo_name;
    let max_depth = Array.length nb_cubes_per_depth - 1 in
    printf "%d\n" max_depth;
    for depth = 1 to max_depth do
        let cubes = read_cubes depth nb_cubes_per_depth.(depth) in
        let data = get_path_len_data f cubes in
        let len = Array.length data in
        printf "%d\n" len;
        for i = 0 to len - 1 do
            printf "%d\n" data.(i);
        done;
    done


(* ----- MAIN ----- *)

let () =
    (* Printexc.record_backtrace true; *)
    Out_channel.set_buffered stdout false;  (* False: disable buffered mode *)
    Random.self_init ();

    let scale_lin min max max_depth =
        let alpha = (min - max) / (max_depth - 1) in
        Array.init (max_depth + 1) (fun x -> if x = 0 then 0 else alpha * (x - 1) + max)
    in
    let scale_exp min max max_depth =
        let min_f, max_f = float_of_int min, float_of_int max in
        let max_depth_f = float_of_int max_depth in
        let alpha = (log min_f -. log max_f) /. (max_depth_f -. 1.) in
        let f x =
            if x = 0. then 0.
            else exp (alpha *. (x -. 1.) +. log max_f)
        in
        Array.init (max_depth + 1) (fun x -> int_of_float (f (float_of_int x)))
    in

    (* let cube = random_cube 5 in
    print_moves_list (a_star_solver cube);
    print_moves_list (korf cube);
    print_moves_list (bfs cube moves_list); *)

    (* let cubes = full_random_cubes 100_000 in
    let f = thistlethwaite_ida_star in
    let algo_name = "Thistlethwaite (avec IDA* )" in
    let data = get_path_len_data f cubes in
    let len = Array.length data in
    printf "%s\n" algo_name;
    printf "%d\n" len;
    for i = 0 to len - 1 do
        printf "%d\n" data.(i);
    done;
    let f = kociemba_ida_star in
    let algo_name = "Kociemba (avec IDA* )" in
    let data = get_path_len_data f cubes in
    let len = Array.length data in
    printf "%s\n" algo_name;
    printf "%d\n" len;
    for i = 0 to len - 1 do
        printf "%d\n" data.(i);
    done; *)

    let tmp n =
        match n with
        | 1 ->
            let max_depth = 20 in
            let n = 10_000 in
            let nb_cubes_per_depth = Array.make (max_depth + 1) n in
            get_and_print_time_data thistlethwaite_ida_star nb_cubes_per_depth
                "Thistlethwaite (avec IDA*)";
            unload_thistlethwaite_hashtables ();
            (* get_and_print_data thistlethwaite_bfs max_nb_moves n "thistlethwaite_bfs"; *)
            get_and_print_time_data kociemba_ida_star nb_cubes_per_depth "Kociemba (avec IDA*)";
            unload_kociemba_hashtables ();
        | 2 ->
            let min = 5 in
            let max = min * 10_000 in
            let max_depth = 15 in
            let nb_cubes_per_depth = scale_exp min max max_depth in
            get_and_print_time_data korf nb_cubes_per_depth "Korf (IDA*)";
        | 3 ->
            let min = 2 in
            let max = min * 1_000 in
            let max_depth = 15 in
            let nb_cubes_per_depth = scale_exp min max max_depth in
            nb_cubes_per_depth.(max_depth) <- 0;
            get_and_print_time_data a_star_solver nb_cubes_per_depth "A*";
            unload_korf_hashtables ();
        | 4 ->
            let min = 10 in
            let max = min * 10_000 in
            let max_depth = 5 in
            let nb_cubes_per_depth = scale_exp min max max_depth in
            get_and_print_time_data bfs nb_cubes_per_depth "BFS Naif";
        | 5 ->
            let max_depth = 20 in
            let n = 10_000 in
            let nb_cubes_per_depth = Array.make (max_depth + 1) n in
            get_and_print_path_len_data thistlethwaite_ida_star nb_cubes_per_depth
                "Thistlethwaite (avec IDA*)";
            unload_thistlethwaite_hashtables ();
            (* get_and_print_data thistlethwaite_bfs max_nb_moves n "thistlethwaite_bfs"; *)
            get_and_print_path_len_data kociemba_ida_star nb_cubes_per_depth "Kociemba (avec IDA*)";
            unload_kociemba_hashtables ();
        | _ -> failwith "tmp"
    in
    generate_cubes (Array.make 21 100_000);
    tmp 4
```

```
(* for i = 1 to 5 do *)
    (* tmp i *)
(* done; *)
(* tmp 4 *)
```