

```

/*
 * representations.c
 *
 * find_representations() finds transitive representations of a manifold's
 * fundamental group into  $S(n)$ , the symmetric group on  $n$  letters.
 * The representations may then be passed to construct_cover() to obtain
 * the corresponding covering spaces. Please see covers.h for details.
 * A representation is "transitive" iff the corresponding covering space
 * is connected.
 *
 * Pass range = permutation_subgroup_Sn to find all transitive
 * representations into  $S(n)$ , up to conjugacy. This yields all  $n$ -fold
 * covering spaces, up to equivalence.
 *
 * Pass range = permutation_subgroup_Zn to find all transitive
 * representations into  $Z/n$ , up to automorphisms of  $Z/n$ . The next
 * section below proves that this yields all  $n$ -fold cyclic covers,
 * up to equivalence. (Note:  $S(n)$  may contain many  $Z/n$  subgroups.
 * We use the "obvious" one generated by  $i \rightarrow i+1$ .)
 *
 * For large values of  $n$  it's *much* faster to find cyclic representations
 * than all representations, because there are only  $n$  groups elements
 * to be considered for each generator instead of  $n!$ .
 *
 * At some point in the future it may be possible to pass
 * range == permutation_subgroup_Dn to find dihedral coverings,
 * but that's not implemented yet.
 *
 * The cusps may be unfilled or have integer Dehn filling coefficients
 * (non relatively prime integers are OK -- they define orbifolds).
 * However if noninteger coefficients are present the fundamental group
 * isn't well defined, and find_representations() returns an empty list.
 */

/*
 * Cyclic Coverings and Representations in  $Z/n$ 
 *
 * (The documentation at the top of cover.h provides the background
 * necessary to understand this section.)
 *
 * Without loss of generality we may assume that the sheets of a cyclic
 * covering are numbered consecutively, so that some generator of the
 * covering's  $Z/n$  symmetry group takes sheet  $i$  to sheet  $i+1$  (mod  $n$ ).
 * It follows that the representation of the fundamental group in  $S(n)$ 
 * is actually a representation into this  $Z/n$  subgroup. By finding
 * all representations into  $Z/n$ , we find all cyclic coverings.
 * The question is, how do we know which ones are equivalent?
 *
 * First recall the situation for arbitrary representations in  $S(n)$ .
 *
 * Proposition. Two different representations yield equivalent covering
 * spaces iff the representations differ by conjugacy in  $S(n)$ .
 *
 * Proof. Equivalent coverings may differ only in the way the labels
 *  $0, 1, \dots, n-1$  are assigned to the sheets. Q.E.D.
 *
 * The special case of a cyclic covering is a little more delicate.
 * We could apply the above proposition to decide when two cyclic
 * coverings were equivalent, but it would be very slow -- we'd have
 * to check all  $n!$  possible conjugators, and our algorithm would slow
 * to a crawl. Fortunately we can do much better. In fact, our assumption
 * that the sheets of the cover be numbered consecutively implies that
 * we need to check fewer than  $n$  possible conjugators.
 *
 * Proposition. If two connected cyclic coverings (with consecutively
 * numbered sheets, as explained above) are equivalent, they are equivalent
 * by a map of the form  $f(i) = p*i$ , for some  $p$  relatively prime to  $n$ .
 *
 * Proof. Say we have two representations from the fundamental group
 * into  $Z/n$  which yield equivalent cyclic covers. That is, there is some
 * map  $f()$  from the sheets  $\{0, 1, \dots, n-1\}$  of the first cover to the sheets
 *  $\{0, 1, \dots, n-1\}$  of the second cover, and it respects the connections
 * between sheets. Because each cover has  $n$ -fold cyclic symmetry, we may
 * without loss of generality assume that  $f(0) = 0$ . Because the coverings

```

```

* are connected, there is some path in the manifold which lifts to a path
* from sheet 0 to sheet 1 in the first covering. In the second covering
* that same path lifts to a path from sheet 0 to some other sheet p.
* Because f() respects the connections between sheets, this implies that
* f(1) = p. Because the sheets are numbered consecutively, if you
* continue the lift of the path in the first manifold, you'll go from
* sheet 1 to sheet 2. In the second covering you'll go from sheet p
* to sheet 2p. Thus f(2) = 2p. And so on. Hence f(i) = p*i (mod n).
* Furthermore p must be relatively prime to n, since otherwise f()
* wouldn't be one-to-one. Q.E.D.
*/

/*
* Conjugacy and Minimality
*
* To find all representations up to conjugacy we could, in principle,
* find all representations and then check which ones are conjugate
* to which others. But this would be very, very slow. Instead we will
* define a canonical ordering on S(n), and use it to reject representations
* which are not lexicographically minimal relative to this ordering.
*
* Indeed, if we (tentatively) assign a permutation to the first generator
* of a group presentation, and notice that that permutation is not
* a minimal one, we may ignore it and move on to the next possibility.
* In particular, we needn't explicitly go through all the
* (n!)^(num generators - 1) ways of assigning permutations to the
* remaining generators. Most permutations aren't minimal, so this
* speeds our algorithm up enormously.
*
* So how do we define the canonical ordering on S(n), and how do
* we recognize the minimal element in an equivalence class?
* We'll answer the second question before the first.
*
* An element of S(n) is defined up to conjugacy by the structure
* of its cycles, for example
*
*          (x)(x)(xx)(xxx)(xxx)
*
* Define the "minimal" element in the conjugacy class to be
* the one obtained by filling in the symbols 0, 1, ..., n-1
* in order, e.g.
*
*          (0)(1)(23)(456)(789)
*
* Proposition. When written in the form
*
*          0 1 2 3 4 5 6 7 8 9
*      ->  0 1 3 2 5 6 4 8 9 7
*
* this minimal form is lexicographically least among all its conjugates.
*
* Proof. To minimize the first entry, we'd like to map 0 to 0.
* This is possible iff a cyclic of length one is present. Similarly,
* to minimize the second element we'd like to map 1 to 1, which is
* possible iff a second cycle of length one is present. Eventually
* we run out of cycles of length one, and the best we can do is to
* map the symbol i to i+1. For minimality we'd like to map something
* to i as soon as possible, which means being part of a minimal length
* cycle. And so on until we've filled in the whole permutation. Q.E.D.
*
* So we let the canonical ordering on S(n) be the lexicographic ordering.
* The above proposition implies that a minimal element relative to
* this order is of the form, e.g., (0)(1)(23)(456)(789).
*
* The above definitions do not carry over to representations in Z/n.
* For example, the permutation (02)(13) is an element of the standard
* Z/n subgroup of S(n), but its minimal form (01)(23) is not.
* Instead we'll use the natural ordering Z/n already has, namely
* 0, 1, ..., n-1.
*
* Proposition. For each nonzero element k of Z/n, there is an
* automorphism of Z/n taking k to gcd(k,n).
*
* Proof. Let

```

```

*          g  = gcd(k,n)
*          k'  = k/g
*          n'  = n/g
*
* An automorphism of  $\mathbb{Z}/n$  is defined by a map  $f(i) = p*i$ , for some  $p$ 
* relatively prime to  $n$ . For convenience, we'll find an automorphism
* taking  $g$  to  $k$ ; that is, we'll find the inverse of the automorphism
* advertised in the statement of the proposition. To do this, we need
* to find a  $p$  such that
*
*          (1)  $p*g == k \pmod{n}$ 
* and
*          (2)  $p$  and  $n$  are relatively prime.
*
* I claim that  $p = k' + n/g'$  does the job, where  $g'$  is the product
* of all factors of  $g$  (with original multiplicities) which also occur
* (at least once) in  $k'$ . [Note that  $g'$  is not quite the same thing as
* the greatest common divisor of  $g$  and  $k'$ . For example, if  $k' = 2*3 = 6$ 
* and  $g = 3*3*5 = 45$ , then  $g' = 3*3 = 9$  even though  $\gcd(g,k') = 3$ .]
*
* Clearly  $p$  satisfies condition (1):
*
*           $p*g = g*k' + (g/g')*n == k \pmod{n}$ 
*
* To check condition (2), we must check whether  $p = k' + n/g'$  can have
* any prime factors in common with  $n$ . There are three kinds of prime
* factors of  $n$  to consider:
*
* (A) prime factors which occur in  $n$  but not in  $g$ ,
* (B) prime factors which occur in  $g$  but not in  $g'$ ,
* (C) prime factors which occur in  $g'$ .
*
* Type (A) factors divide  $n/g'$ . But they cannot divide  $k'$ , because
* if they did they'd be part of  $g$ . Therefore type (A) factors do not
* divide  $p$ .
*
* Type (B) factors also divide  $n/g'$  but not  $k'$ , so they don't divide  $p$ .
*
* Type (C) factors divide  $k'$  (this is obvious from the definition of  $g'$ )
* but do not divide  $n/g'$  (additional copies of the factor can't
* lie in  $n/g$  because that would make them common to both  $k'$  and  $n'$  and
* therefore part of  $g$ , and can't lie in  $g/g'$  because the definition
* of  $g'$  already includes all copies of each included factor).
*
* Q.E.D.
*
* Corollary. A nonzero element  $k$  of  $\mathbb{Z}/n$  is minimal under automorphisms
* of  $\mathbb{Z}/n$  iff  $k$  divides  $n$ .
*
* Proof.
* ( $\Leftarrow$ ) If  $k$  divides  $n$ , then  $k$  is an element of order  $n/k$ . Lesser
* elements must have higher order, and therefore cannot be equivalent
* to  $k$  under any automorphism.
* ( $\Rightarrow$ ) If  $k$  does not divide  $n$ , then the automorphism provided by the
* preceding proposition takes  $k$  to  $\gcd(k,n)$ , which is less than  $k$ .
* Q.E.D.
*/

/*
* Minor technical comment (which you may ignore).
*
* The algorithm for finding representations into  $\mathbb{Z}/n$  does not
* make use of the fact that  $\mathbb{Z}/n$  is abelian, even though doing so
* might speed it up a tiny bit. I felt it was more important
* to keep the overall structure of the algorithm the same for
* the  $\mathbb{Z}/n$  and  $S(n)$  cases (and perhaps eventually for  $D_n$  if it
* gets added). And in any case, finding representations into
*  $\mathbb{Z}/n$  isn't a bottleneck: creating the final Triangulations
* is much slower than finding the representations to begin with.
*/
#include "kernel.h"

static int          **compute_Sn(int n);
static void          free_Sn(int **Sn, int n_factorial);

```

```

static int factorial(int n);
static Boolean first_indices_all_zero(int *representation_by_index, int
    num_generators);
static Boolean group_element_is_Sn_minimal(int *permutation, int num_sheets);
static Boolean group_element_is_Zn_minimal(int index, int num_sheets);
static Boolean candidateSn_is_valid(int **candidateSn, int n, int **
    group_relations, int num_relations);
static Boolean candidateZn_is_valid(int *candidateZn, int n, int **
    group_relations, int num_relations);
static Boolean candidateSn_is_transitive(int **candidateSn, int num_generators
    , int n);
static Boolean candidateZn_is_transitive(int *candidateZn, int num_generators
    , int n);
static Boolean candidateSn_is_conjugacy_minimal(int **candidateSn, int
    num_generators, int n, int **Sn, int n_factorial);
static Boolean candidateZn_is_conjugacy_minimal(int *candidateZn, int
    num_generators, int n);
static RepresentationIntoSn *convert_candidateSn_to_original_generators(int **candidateSn,
    int n, int num_original_generators, int **original_generators, Triangulation *manifold,
    int **meridians, int **longitudes);
static RepresentationIntoSn *convert_candidateZn_to_original_generators(int *candidateZn,
    int n, int num_original_generators, int **original_generators, Triangulation *manifold,
    int **meridians, int **longitudes);
static RepresentationIntoSn *initialize_new_representation(int num_original_generators, int
    n, int num_cusps);
static void word_to_Sn(int **candidateSn, int *word, int *permutation, int
    n);
static int word_to_Zn(int *candidateZn, int *word, int n);
static void compute_primitive_Dehn_coefficients(Cusp *cusp, int *
    primitive_m, int *primitive_l);
static void compose_with_power(int *product, int *factor, int power, int n);
static void Zn_to_Sn(int element_of_Zn, int *element_of_Sn, int n);
static void compute_covering_type(RepresentationIntoSn *representation, int
    num_generators, int n);
static void free_representation(RepresentationIntoSn *representation, int
    num_generators, int num_cusps);

```

```

RepresentationList *find_representations(
    Triangulation *manifold,
    int n,
    PermutationSubgroup range)
{
    RepresentationList *representation_list;
    int i,
        n_factorial,
        range_size,
        num_simplified_generators,
        num_simplified_relations,
        num_original_generators,
        **simplified_group_relations,
        **original_generators,
        **meridians,
        **longitudes,
        digit,
        **Sn,
        *representation_by_index,
        **candidateSn,
        *candidateZn;
    GroupPresentation *simplified_group;
    RepresentationIntoSn *new_representation;

    /*
     * Begin with a quick error check.
     */
    if (manifold == NULL)
        uFatalError("find_representations", "representations");

    /*
     * Make sure the manifold has a set of generators.
     */
    choose_generators(manifold, FALSE, FALSE);
}

```

```

/*
 * Allocate the RepresentationList.
 */
representation_list = NEW_STRUCT(RepresentationList);
representation_list->num_generators = manifold->num_generators;
representation_list->num_sheets = n;
representation_list->num_cusps = manifold->num_cusps;
representation_list->list = NULL;

/*
 * The cusps must be unfilled or have integer Dehn filling
 * coefficients (non relatively prime integers are OK -- they
 * define orbifolds). Otherwise we can't define a representation
 * into  $S(n)$  in any meaningful way, so we return an empty list.
 */
if (all_De hn_coefficients_are_integers(manifold) == FALSE)
    return representation_list;

/*
 * If n is less than 1, return an empty list.
 */
if (n < 1)
    return representation_list;

/*
 * Get a simplified presentation for the fundamental group.
 * Use minimize_number_of_generators == TRUE, because the number
 * of (potential) representations is an exponential function
 * of the number of generators. At the end we'll convert
 * the final representations to the standard generators defined in
 * choose_generators().
 *
 * 97/4/7 If the group is trivial, return an empty list.
 */
simplified_group = fundamental_group(manifold, TRUE, TRUE, TRUE);
num_simplified_generators = fg_get_num_generators(simplified_group);
if (num_simplified_generators == 0)
{
    free_group_presentation(simplified_group);
    return representation_list;
}
num_simplified_relations = fg_get_num_relations(simplified_group);
simplified_group_relations = NEW_ARRAY(num_simplified_relations, int *);
for (i = 0; i < num_simplified_relations; i++)
    simplified_group_relations[i] = fg_get_relation(simplified_group, i);
num_original_generators = manifold->num_generators;
original_generators = NEW_ARRAY(num_original_generators, int *);
for (i = 0; i < num_original_generators; i++)
    original_generators[i] = fg_get_original_generator(simplified_group, i);
meridians = NEW_ARRAY(manifold->num_cusps, int *);
longitudes = NEW_ARRAY(manifold->num_cusps, int *);
for (i = 0; i < manifold->num_cusps; i++)
{
    meridians[i] = fg_get_meridian(simplified_group, i);
    longitudes[i] = fg_get_longitude(simplified_group, i);
}
free_group_presentation(simplified_group);

/*
 * If the range is all of  $S(n)$ , it will be convenient to precompute
 * an array containing all its elements.
 */
if (range == permutation_subgroup_Sn)
{
    n_factorial = factorial(n);
    Sn = compute_Sn(n);
}
else
{
    n_factorial = 0;
    Sn = NULL;
}

/*

```

```

    * If the range is all of  $S(n)$ , then each "candidate" representation
    * in the loop below will be expressed as an array of pointers
    * to rows of the array  $S_n$ , one pointer for each generator.
    */
if (range == permutation_subgroup_Sn)
    candidateSn = NEW_ARRAY(num_simplified_generators, int *);
else
    candidateSn = NULL;

/*
 * The representation_by_index array keeps track of the index
 * of the group element assigned to each generator. If the range
 * is all of  $S(n)$ , the indices 0, 1, ..., ( $n\_factorial - 1$ ) refer
 * to the permutations  $S_n[0]$ ,  $S_n[1]$ , ...,  $S_n[n\_factorial - 1]$ .
 * If the range is  $Z/n$ , the indices 0, 1, ...,  $n-1$  refer naturally
 * to the corresponding elements of  $Z/n$ .
 *
 * N.B. The extra entry at the end of the representation_by_index
 * array is used to detect when the enumeration is complete.
 */
representation_by_index = NEW_ARRAY(num_simplified_generators + 1, int);
if (range == permutation_subgroup_Sn)
    range_size = n_factorial;
else
    range_size = n;

/*
 * Initialize all representation indices to 0.
 */
for (i = 0; i < num_simplified_generators + 1; i++)
    representation_by_index[i] = 0;

/*
 * Loop through all possible assignments of range elements to generators.
 * Roughly speaking, this is just like counting base range_size, with
 * the least significant digits at the left. For example, if  $n = 3$ ,
 *  $range\_size = 3! = 6$ , and  $num\_simplified\_generators = 4$ , the
 * enumeration would be
 *
 *      00000
 *      10000
 *      20000
 *      30000
 *      40000
 *      50000
 *      01000
 *      11000
 *      21000
 *      ...
 *      35550
 *      45550
 *      55550
 *      00001  <-- the trailing 1 means were done
 */
while (representation_by_index[num_simplified_generators] == 0) /* Loop until we reach 00001 */
{
    /*
     * If the range is all of  $S(n)$ , convert the representation_by_index[]
     * to an array of pointers to rows of  $S_n$ . Otherwise define
     * candidateZn to be a synonym for the representation_by_index[]
     * array itself.
     */
    if (range == permutation_subgroup_Sn)
        for (i = 0; i < num_simplified_generators; i++)
            candidateSn[i] = Sn[representation_by_index[i]];
    else
        candidateZn = representation_by_index;

    /*
     * Every representation is conjugate to one in which the high-order
     * permutation, i.e.
     *
     *      representation_by_index[num_simplified_generators - 1])
    */
}

```

```

    *
    * is minimal (as defined in the section Conjugacy and Minimality
    * at the top of this file), so we may safely ignore representations
    * in which the high-order permutation isn't minimal. In practice,
    * this means that whenever the first (num_simplified_generators - 1)
    * permutation indices are zero, we check the last index, and if
    * it's not a minimal permutation we increment the index immediately.
    */
    if
    ( first_indices_all_zero(representation_by_index, num_simplified_generators) ==
TRUE
    && (
        range == permutation_subgroup_Sn ?
        group_element_is_Sn_minimal(candidateSn[num_simplified_generators - 1], n)
    == FALSE :
        group_element_is_Zn_minimal(candidateZn[num_simplified_generators - 1], n)
    == FALSE
    )
    )
    {
        /*
        * Increment the high order index, and check for a carry.
        */
        representation_by_index[num_simplified_generators - 1]++;
        if (representation_by_index[num_simplified_generators - 1] == range_size)
        {
            representation_by_index[num_simplified_generators - 1] = 0;
            representation_by_index[num_simplified_generators] = 1;
        }
        /*
        * Continue with the while() loop.
        */
        continue;
    }

    /*
    * Add the candidate to the representation_list iff it
    *
    * (1) respects the group relations,
    * (2) is transitive, and
    * (3) is minimal under conjugacy.
    *
    * The precise meaning of condition 3 is explained in
    * candidateSn_is_conjugacy_minimal().
    *
    * Note: A quick experiment showed that the order of the calls
    * to candidateSn_is_valid() and candidateSn_is_transitive() has
    * no consistent effect on the run time, but the call to
    * candidateSn_is_conjugacy_minimal() should definitely be last.
    */
    if (range == permutation_subgroup_Sn ?
        ( candidateSn_is_valid(candidateSn, n, simplified_group_relations,
num_simplified_relations)
        && candidateSn_is_transitive(candidateSn, num_simplified_generators, n)
        && candidateSn_is_conjugacy_minimal(candidateSn, num_simplified_generators, n,
Sn, n_factorial)
        ) :
        ( candidateZn_is_valid(candidateZn, n, simplified_group_relations,
num_simplified_relations)
        && candidateZn_is_transitive(candidateZn, num_simplified_generators, n)
        && candidateZn_is_conjugacy_minimal(candidateZn, num_simplified_generators, n)
        )
    )
    {
        new_representation = range == permutation_subgroup_Sn ?
                                convert_candidateSn_to_original_generators
(candidateSn, n, num_original_generators, original_generators, manifold, meridians,
longitudes) :
                                convert_candidateZn_to_original_generators
(candidateZn, n, num_original_generators, original_generators, manifold, meridians,
longitudes);
        new_representation->next = representation_list->list;
        representation_list->list = new_representation;
    }
}

```

```

    /*
     * Increment the representation_by_index base range_size
     * as described above.
     *
     * (As a special case, if n == 1 just break, since the base 1
     * arithmetic won't work.)
     */
    if (n == 1)
        break;
    digit = 0;          /* start at the low-order digit */
    while (TRUE)        /* loop until we break */
    {
        representation_by_index[digit]++;          /* increment current digit */
        /*
         * if (representation_by_index[digit] == range_size) /* is there a carry?
         */
        {
            representation_by_index[digit] = 0;    /* set this digit to 0
            */
            digit++;                                /* prepare to increment
            */
            next digit /*
            */
        }
        else
            break;                                  /* no carry, so we're done
            */
    }
}

/*
 * Free local arrays.
 */

if (range == permutation_subgroup_Sn)
{
    free_Sn(Sn, n_factorial);
    my_free(candidateSn);
}
my_free(representation_by_index);

for (i = 0; i < num_simplified_relations; i++)
    fg_free_relation(simplified_group_relations[i]);
my_free(simplified_group_relations);

for (i = 0; i < num_original_generators; i++)
    fg_free_relation(original_generators[i]);
my_free(original_generators);

for (i = 0; i < manifold->num_cusps; i++)
{
    fg_free_relation(meridians[i]);
    fg_free_relation(longitudes[i]);
}
my_free(meridians);
my_free(longitudes);

/*
 * Done!
 */
return representation_list;
}

static int **compute_Sn(
    int n)
{
    /*
     * We want to enumerate the elements of S(n) in a systematic way.
     * For concreteness, consider the problem of enumerating the
     * elements of S(4). To ensure that each symbol {0,1,2,3}
     * appears in the last position, we cyclically advance the
     * symbols four times:
     *
     *      0123
    */

```



```
*          3012
*          2301
*          1230
*
* For each fixed choice for the last position, we repeat
* the process for the "subword" consisting of the first
* three positions:
*
*          0123
*          0123
*          2013
*          1203
*          3012
*          3012
*          1302
*          0132
*          2301
*          2301
*          0231
*          3021
*          1230
*          1230
*          3120
*          2130
*
* With the last two positions fixed, permute the remaining
* subword, etc.
*
*          0123
*          0123
*          0123
*          1023
*          2013
*          2013
*          2013
*          0213
*          etc.
*          etc.
*
* The rule for obtaining the final list
*
*          0123
*          1023
*          2013
*          0213
*          1203
*          2103
*          3012
*          0312
*          1302
*          etc.
*          0123
*          1023
*          2013
*          0213
*          1203
*          2103
*          3012
*          0312
*          1302
*          3102
*          0132
*          1032
*          2301
*          3201
*          0231
*          2031
*          3021
*          0321
*          1230
*          2130
*          3120
*          1320
*          2310
*          3210
```

```

*
* is that permutation 0 is the identity, and thereafter permutation i
* is obtained from permutation i-1 by
*
*     cycling the first 2 entries iff i is divisible by 1!, and
*     cycling the first 3 entries iff i is divisible by 2!, and
*     cycling the first 4 entries iff i is divisible by 3!.
*
* Note that the above clauses are joined by "and", so that, e.g.
* for i == 12 you'd first cycle the first 2 entries, then the
* first 3, and then all 4 to obtain the correct final permutation.
*/

int **Sn,
    n_factorial,
    i,
    j,
    k,
    subword_length,
    required_divisor,
    temp;

/*
 * Compute n_factorial.
 */
n_factorial = factorial(n);

/*
 * Allocate space for the array of permutations.
 */
Sn = NEW_ARRAY(n_factorial, int *);
for (i = 0; i < n_factorial; i++)
    Sn[i] = NEW_ARRAY(n, int);

/*
 * Sn[0] will be the identity permutation.
 */
for (j = 0; j < n; j++)
    Sn[0][j] = j;

/*
 * All subsequent permutations are defined recursively,
 * by cycling subwords as explained above.
 */
for (i = 1; i < n_factorial; i++)
{
    /*
     * Copy Sn[i-1] into Sn[i].
     */
    for (j = 0; j < n; j++)
        Sn[i][j] = Sn[i-1][j];

    /*
     * Cycle an initial subword iff (subword_length - 1)! divides i.
     * (Note that shorter subwords cycle before longer ones, to
     * return the longer subword to its previous condition.)
     */
    for (
        subword_length = 2, required_divisor = 1;
        subword_length <= n;
        required_divisor *= subword_length, subword_length++)
    {
        if (i % required_divisor == 0)
        {
            temp = Sn[i][subword_length - 1];
            for (k = subword_length - 1; k > 0; --k)
                Sn[i][k] = Sn[i][k-1];
            Sn[i][0] = temp;
        }
    }
}

return Sn;
}

```

```

static void free_Sn(
    int **Sn,
    int n_factorial)
{
    int i;

    for (i = 0; i < n_factorial; i++)
        my_free(Sn[i]);
    my_free(Sn);
}

static int factorial(
    int n)
{
    int n_factorial;

    n_factorial = 1;
    while (n > 0)
    {
        n_factorial *= n;
        --n;
    }

    return n_factorial;
}

static Boolean first_indices_all_zero(
    int *representation_by_index,
    int num_generators)
{
    int i;

    for (i = 0; i < num_generators - 1; i++)
        if (representation_by_index[i] != 0)
            return FALSE;

    return TRUE;
}

static Boolean group_element_is_Sn_minimal(
    int *permutation,
    int num_sheets)
{
    /*
     * Is the permutation minimal, as defined in find_representations()?
     * Recall that an example of a minimal permutation is
     *
     *             (0)(1)(23)(456)(789)
     *
     * or, in another form,
     *
     *             0 1 2 3 4 5 6 7 8 9
     *             -> 0 1 3 2 5 6 4 8 9 7
     *
     * To test whether a permutation is minimal, we must check that
     * (1) cycles consist of consecutive integers, and
     * (2) the cycles lengths occur in ascending order.
     */

    int position,
        cycle_start,
        old_cycle_length,
        new_cycle_length;

    position = 0;
    cycle_start = 0;
    old_cycle_length = 0;

    while (position < num_sheets)
    {

```

```

        if (permutation[position] == position + 1)          /* cycle continues */
        {
            position++;
        }
        else if (permutation[position] == cycle_start) /* cycle completes */
        {
            new_cycle_length = (position - cycle_start) + 1;
            if (new_cycle_length < old_cycle_length)
                return FALSE;
            old_cycle_length = new_cycle_length;
            position++;
            cycle_start = position;
        }
        else /* bad cycle */
            return FALSE;
    }

    return TRUE;
}

static Boolean group_element_is_Zn_minimal(
    int index,
    int num_sheets)
{
    if (index == 0)

        /*
         * The identity element of  $Z/n$  is always minimal,
         * because it's fixed by all automorphisms.
         */
        return TRUE;

    else

        /* By a corollary in the documentation at the top of this file,
         * a nonzero element  $k$  of  $Z/n$  is minimal under automorphisms
         * of  $Z/n$  iff  $k$  divides  $n$ .
         */
        return (num_sheets % index == 0);
}

static Boolean candidateSn_is_valid(
    int **candidateSn,
    int n,
    int **group_relations,
    int num_relations)
{
    Boolean satisfies_all_relations;
    int *permutation,
        i,
        j;

    /*
     * Initialize satisfies_all_relations to TRUE.
     */
    satisfies_all_relations = TRUE;

    /*
     * Allocate scratch space to hold permutation which the
     * candidate assigns to a given group relation.
     */
    permutation = NEW_ARRAY(n, int);

    /*
     * Check each relation.
     */
    for (i = 0; i < num_relations; i++)
    {
        /*
         * Compute the permutation which the candidate assigns
         * to this group relation.
         */
    }

```

```
    word_to_Sn(candidateSn, group_relations[i], permutation, n);

    /*
     * The relation is satisfied iff the permutation is the identity.
     */
    for (j = 0; j < n; j++)
        if (permutation[j] != j)
            satisfies_all_relations = FALSE;

    /*
     * If this relation isn't satisfied, there is no point
     * in looking at the remaining ones.
     */
    if (satisfies_all_relations == FALSE)
        break;
}

/*
 * Free the scratch space.
 */
my_free(permutation);

/*
 * All done.
 */
return satisfies_all_relations;
}

static Boolean candidateZn_is_valid(
    int      *candidateZn,
    int      n,
    int      **group_relations,
    int      num_relations)
{
    Boolean  satisfies_all_relations;
    int      group_element,
            i;

    /*
     * Initialize satisfies_all_relations to TRUE.
     */
    satisfies_all_relations = TRUE;

    /*
     * Check each relation.
     */
    for (i = 0; i < num_relations; i++)
    {
        /*
         * Compute the permutation which the candidate assigns
         * to this group relation.
         */
        group_element = word_to_Zn(candidateZn, group_relations[i], n);

        /*
         * The relation is satisfied iff the group_element
         * is the identity element of Z/n.
         */
        if (group_element != 0)
            satisfies_all_relations = FALSE;

        /*
         * If this relation isn't satisfied, there is no point
         * in looking at the remaining ones.
         */
        if (satisfies_all_relations == FALSE)
            break;
    }

    /*
     * All done.
     */
    return satisfies_all_relations;
}
```

```

}

static Boolean candidateSn_is_transitive(
    int      **candidateSn,
    int      num_generators,
    int      n)
{
    Boolean   *visited,
              progress;
    int       count,
              i,
              j;

    /*
     * Is the candidate transitive?
     * In other words, is the corresponding covering space connected?
     *
     * Start on sheet 0 of the cover and see which other sheets we can
     * get to using the permutations assigned to the generators.
     * A "visited" array will keep track of which sheets we can reach.
     */
    visited = NEW_ARRAY(n, Boolean);
    for (i = 0; i < n; i++)
        visited[i] = FALSE;

    /*
     * Begin by visiting sheet 0.
     */
    visited[0] = TRUE;
    count      = 1;

    /*
     * See which other sheets we can visit by following generators of the
     * fundamental group. Keep going as long as we're making progress.
     */
    do
    {
        progress = FALSE;

        for (i = 0; i < num_generators; i++)
            for (j = 0; j < n; j++)
                if (visited[j])
                    if ( ! visited[candidateSn[i][j]])
                    {
                        visited[candidateSn[i][j]] = TRUE;
                        count++;
                        progress = TRUE;
                    }

    } while (progress == TRUE);

    /*
     * Free the visited array.
     */
    my_free(visited);

    /*
     * Return TRUE iff we visited all the sheets.
     */
    return (count == n);
}

static Boolean candidateZn_is_transitive(
    int      *candidateZn,
    int      num_generators,
    int      n)
{
    /*
     * The candidate will be transitive iff the greatest common divisor
     * of the group elements is 1.
     */

```

```

    int     the_gcd,
            i;

    the_gcd = n;

    for (i = 0; i < num_generators; i++)
        the_gcd = gcd(the_gcd, candidateZn[i]);

    return (the_gcd == 1);
}

static Boolean candidateSn_is_conjugacy_minimal(
    int     **candidateSn,
    int     num_generators,
    int     n,
    int     **Sn,
    int     n_factorial)
{
    /*
     * Define the lexicographic ordering on S(n) to be, e.g.,
     *
     *      0123
     *      0132
     *      0213
     *      0231
     *      0312
     *      0321
     *      1023
     *      1032
     *      etc.
     *
     * That is, the symbol in the first is most important, and the
     * symbol in the last position is least important.
     *
     * Warnings:
     *
     * (1) Don't confuse this with the lexicographic ordering on the
     * representations, in which the last generator is most important,
     * and the first generator is least important!
     *
     * (2) This lexicographic ordering on S(n) is NOT the order
     * in which the elements of the array Sn appear!
     *
     * We want to check whether the given candidate is minimal among all
     * its conjugates. To compare the candidate to one of its conjugates,
     * we will need to use BOTH the right-to-left ordering on the set
     * of all representations, and the left-to-right ordering on S(n).
     * Clear?!?
     */

    int i,
        j,
        k,
        *scratch,
        comparison;

    /*
     * Allocate scratch space to compute the conjugate.
     */
    scratch = NEW_ARRAY(n, int);

    /*
     * Consider each of the possible n! conjugating permutations.
     */
    for (i = 0; i < n_factorial; i++)
    {
        /*
         * Consider the permutation assigned to each generator,
         * starting with the last one and working backwards.
         */
        for (j = num_generators; --j >= 0; )
        {
            /*

```

```

    * Copy the inverse of Sn[i] into scratch.
    */
    for (k = 0; k < n; k++)
        scratch[Sn[i][k]] = k;

    /*
    * Multiply by the permutation assigned to generator j.
    */
    for (k = 0; k < n; k++)
        scratch[k] = candidateSn[j][scratch[k]];

    /*
    * Multiply by Sn[i].
    */
    for (k = 0; k < n; k++)
        scratch[k] = Sn[i][scratch[k]];

    /*
    * Scratch now holds
    *
    *       $Sn[i]^{-1} * candidateSn[j] * Sn[i]$ 
    *
    * Is it less than, equal to, or greater than
    * candidateSn[j] itself?
    */
    for (k = 0; k < n; k++)
    {
        if (scratch[k] < candidateSn[j][k])
        {
            comparison = -1;
            break;
        }
        else if (scratch[k] > candidateSn[j][k])
        {
            comparison = +1;
            break;
        }
        else
        {
            comparison = 0;
            continue;
        }
    }

    /*
    * If comparison == -1, we've found a conjugate which
    * is less than the original representation, so the
    * original representation can't be minimal.
    */
    if (comparison == -1)
    {
        my_free(scratch);
        return FALSE;
    }

    /*
    * If comparison == +1, then this conjugate is greater
    * than the original representation. We should break
    * out of the j-loop and move on to the next conjugator.
    */
    if (comparison == +1)
        break;

    /*
    * If comparison == 0, then we must continue with the
    * j-loop to check the effect of the current conjugator
    * on the next generator.
    */
}

/*
* The representation has no conjugates lexicographically less
* than itself, so return TRUE.
*/

```



```

    */
    my_free(scratch);
    return TRUE;
}

static Boolean candidateZn_is_conjugacy_minimal(
    int      *candidateZn,
    int      num_generators,
    int      n)
{
    /*
     * The representations are ordered lexicographically, but in
     * a backwards sort of way: the last generator is most important,
     * and the first generator is least important!
     *
     * Relative to this ordering, we want to check whether the given
     * candidate is minimal among all its conjugates.
     */

    int p,
        i,
        image;

    /*
     * According to the section Cyclic Coverings and Representations in Z/n
     * at the top of this file, if two connected cyclic coverings are
     * equivalent, they are equivalent by a map of the form  $f(i) = p \cdot i$ ,
     * for some  $p$  relatively prime to  $n$ . Consider each of the possible
     * values for  $p$ , and test whether any produce an equivalent covering
     * which is lexicographically less than this one.
     */
    for (p = 2; p < n; p++)
    {
        /*
         *  $p$  must be relative prime to  $n$ , or we don't have an automorphism.
         */
        if (gcd(p,n) != 1)
            continue;

        /*
         * Consider the group element assigned to each generator,
         * starting with the last one and working backwards.
         */
        for (i = num_generators; --i >= 0; )
        {
            /*
             * Where does the automorphism take this generator?
             */
            image = (p * candidateZn[i]) % n;

            /*
             * If the image is less than candidateZn[i], we've found
             * a conjugate which is less than the original representation,
             * so the original representation can't be minimal.
             */
            if (image < candidateZn[i])
                return FALSE;

            /*
             * If the image is greater than candidateZn[i], then this
             * conjugate is greater than the original representation.
             * We should break out of the i-loop and move on to the
             * next value of  $p$ .
             */
            if (image > candidateZn[i])
                break;

            /*
             * If the image equals candidateZn[i], then we must continue
             * with the i-loop to check the effect of the current
             * automorphism on the next generator.
             */
        }
    }
}

```

```

    }

    /*
     * The representation has no conjugates lexicographically less
     * than itself, so return TRUE.
     */
    return TRUE;
}

static RepresentationIntoSn *convert_candidateSn_to_original_generators(
    int          **candidateSn,
    int          n,
    int          num_original_generators,
    int          **original_generators,
    Triangulation *manifold,
    int          **meridians,
    int          **longitudes)
{
    /*
     * Convert the candidate from the simplified generators to the
     * original generators, and return it as a RepresentationIntoSn.
     */

    RepresentationIntoSn *representation;
    int i, j,
        meridional_permutation,
        longitudinal_permutation,
        primitive_m,
        primitive_l;

    /*
     * Allocate and initialize the RepresentationIntoSn data structure.
     */
    representation = initialize_new_representation(num_original_generators, n, manifold->
num_cusps);

    /*
     * Allocate scratch space.
     */
    meridional_permutation = NEW_ARRAY(n, int);
    longitudinal_permutation = NEW_ARRAY(n, int);

    /*
     * Compute the permutation assigned to each original generator.
     */
    for (i = 0; i < num_original_generators; i++)
        word_to_Sn(candidateSn, original_generators[i], representation->image[i], n);

    /*
     * Compute the permutation assigned to each Dehn filling curve.
     */
    for (i = 0; i < manifold->num_cusps; i++)
    {
        /*
         * Initialize representation->primitive_Dehn_image[i] to the identity.
         */
        for (j = 0; j < n; j++)
            representation->primitive_Dehn_image[i][j] = j;

        /*
         * If the cusp is unfilled, its primitive_Dehn_image should
         * remain the identity.
         */
        if (find_cusp(manifold, i)->is_complete == TRUE)
            continue;

        /*
         * Compute the permutations assigned to the meridian and longitude.
         */
        word_to_Sn(candidateSn, meridians[i], meridional_permutation, n);
        word_to_Sn(candidateSn, longitudes[i], longitudinal_permutation, n);
    }
}

```

```

    /*
     * Figure out the coefficients of the primitive Dehn filling curve.
     */
    compute_primitive_Deahn_coefficients(find_cusp(manifold, i), &primitive_m, &
    primitive_l);

    /*
     * Compute
     *      meridional_permutation^primitive_m
     *      * longitudinal_permutation^primitive_l
     */
    compose_with_power(representation->primitive_Deahn_image[i], meridional_permutation,
    primitive_m, n);
    compose_with_power(representation->primitive_Deahn_image[i],
    longitudinal_permutation, primitive_l, n);
}

/*
 * Free scratch space.
 */
my_free(meridional_permutation);
my_free(longitudinal_permutation);

/*
 * Does this representation define an irregular, regular or cyclic cover?
 */
compute_covering_type(representation, num_original_generators, n);

/*
 * All done.
 */
return representation;
}

static RepresentationIntoSn *convert_candidateZn_to_original_generators(
int      *candidateZn,
int      n,
int      num_original_generators,
int      **original_generators,
Triangulation *manifold,
int      **meridians,
int      **longitudes)
{
    /*
     * Convert the candidate from the simplified generators to the
     * original generators, and return it as a RepresentationIntoSn.
     *
     * This is also the point where we convert from abstract elements
     * {0, 1, ..., n-1} of Z/n to the corresponding permutations in S(n).
     */

    RepresentationIntoSn *representation;
    int i,
        meridional_transformation,
        longitudinal_transformation,
        primitive_m,
        primitive_l;

    /*
     * Allocate and initialize the RepresentationIntoSn data structure.
     */
    representation = initialize_new_representation(num_original_generators, n, manifold->
    num_cusps);

    /*
     * Compute the element of Z/n assigned to each original generator,
     * and convert it to a full-fledged permutation.
     */
    for (i = 0; i < num_original_generators; i++)
        Zn_to_Sn( word_to_Zn(candidateZn, original_generators[i], n),
        representation->image[i],
        n);
}

```

```

/*
 * Compute the permutation assigned to each Dehn filling curve.
 */
for (i = 0; i < manifold->num_cusps; i++)
{
    /*
     * If the cusp is unfilled, its primitive_Dehn_image should
     * be the identity.
     */
    if (find_cusp(manifold, i)->is_complete == TRUE)
    {
        Zn_to_Sn(0, representation->primitive_Dehn_image[i], n);
        continue;
    }

    /*
     * Compute the group elements assigned to the meridian and longitude.
     */
    meridional_transformation = word_to_Zn(candidateZn, meridians[i], n);
    longitudinal_transformation = word_to_Zn(candidateZn, longitudes[i], n);

    /*
     * Figure out the coefficients of the primitive Dehn filling curve.
     */
    compute_primitive_Dehn_coefficients(find_cusp(manifold, i), &primitive_m, &
    primitive_l);

    /*
     * Compute
     *
     *      primitive_m * meridional_transformation
     *    + primitive_l * longitudinal_transformation
     *
     * and convert it to a full-fledged permutation.
     */
    Zn_to_Sn( ( primitive_m * meridional_transformation
                + primitive_l * longitudinal_transformation),
              representation->primitive_Dehn_image[i],
              n);
}

/*
 * A representation into  $\mathbb{Z}/n$  defines a cyclic covering.
 */
representation->covering_type = cyclic_cover;

/*
 * All done.
 */
return representation;
}

```

```

static RepresentationIntoSn *initialize_new_representation(
    int num_original_generators,
    int n,
    int num_cusps)
{
    RepresentationIntoSn *representation;
    int i;

    representation = NEW_STRUCT(RepresentationIntoSn);

    representation->image = NEW_ARRAY(num_original_generators, int *);
    for (i = 0; i < num_original_generators; i++)
        representation->image[i] = NEW_ARRAY(n, int);

    representation->primitive_Dehn_image = NEW_ARRAY(num_cusps, int *);
    for (i = 0; i < num_cusps; i++)
        representation->primitive_Dehn_image[i] = NEW_ARRAY(n, int);

    representation->covering_type = unknown_cover;

    representation->next = NULL;
}

```

```

    return representation;
}

static void word_to_Sn(
    int      **candidateSn,
    int      *word,
    int      *permutation,
    int      n)
{
    /*
     * Compute the permutation which candidate assigns to the given word.
     */

    int i,
        j,
        gen,
        *factor;

    /*
     * Allocate scratch space.
     */
    factor = NEW_ARRAY(n, int);

    /*
     * Initialize the permutation to the identity.
     */
    for (i = 0; i < n; i++)
        permutation[i] = i;

    /*
     * The word consists of a product of simplified generators.
     * Compose the corresponding permutations.
     */
    for (i = 0; word[i] != 0; i++)
    {
        /*
         * Let the array "factor" hold the permutation which the
         * candidate assigns to the letter word[i]. Recall that
         * word[i] takes on the positive values +1, +2, ... for
         * positive generators and the negative values -1, -2, ...
         * for inverses of generators.
         */
        if (word[i] > 0) /* positive generator */
        {
            /*
             * Copy the permutation directly.
             */
            gen = word[i] - 1;
            for (j = 0; j < n; j++)
                factor[j] = candidateSn[gen][j];
        }
        else /* negative generator */
        {
            /*
             * Compute the inverse permutation.
             */
            gen = (-word[i]) - 1;
            for (j = 0; j < n; j++)
                factor[candidateSn[gen][j]] = j;
        }

        /*
         * Apply the factor to the permutation.
         */
        for (j = 0; j < n; j++)
            permutation[j] = factor[permutation[j]];
    }

    /*
     * Free scratch space.
     */
    my_free(factor);
}

```

```

}

static int word_to_Zn(
    int      *candidateZn,
    int      *word,
    int      n)
{
    /*
     * Compute the element of Z/n which candidate assigns to the given word.
     */

    int i,
        gen,
        sum;

    /*
     * Initialize the sum to the identity.
     */
    sum = 0;

    /*
     * The word consists of a product of simplified generators.
     * Sum the corresponding group elements.
     */
    for (i = 0; word[i] != 0; i++)
    {
        /*
         * Add in the group element which the candidate assigns to the
         * letter word[i]. Recall that word[i] takes on the positive
         * values +1, +2, ... for positive generators and the negative
         * values -1, -2, ... for inverses of generators.
         */
        if (word[i] > 0) /* positive generator */
        {
            gen = word[i] - 1;
            sum += candidateZn[gen];
        }
        else /* negative generator */
        {
            gen = (-word[i]) - 1;
            sum -= candidateZn[gen];
        }
    }

    /*
     * Normalize the result to the range [0, n-1].
     */
    sum = sum % n;
    if (sum < 0) /* The ANSI operator % is not well defined. */
        sum += n; /* If it's negative, add in another +n. */

    return sum;
}

static void compute_primitive_Dehn_coefficients(
    Cusp      *cusp,
    int      *primitive_m,
    int      *primitive_l)
{
    int      the_gcd;

    /* find_representations() has already checked that */
    /* the Dehn filling coefficients are integers. */

    the_gcd = gcd((long int)cusp->m, (long int)cusp->l);

    *primitive_m = (int)cusp->m / the_gcd;
    *primitive_l = (int)cusp->l / the_gcd;
}

static void compose_with_power(

```

```

    int *product,
    int *factor,
    int power,
    int n)
{
    int positive_power,
        *positive_factor,
        i;

    /*
     * Allocate scratch space.
     */
    positive_factor = NEW_ARRAY(n, int);

    /*
     * If power > 0, use the factor as given.
     * If power < 0, use its inverse.
     */
    if (power > 0)
    {
        for (i = 0; i < n; i++)
            positive_factor[i] = factor[i];
        positive_power = power;
    }
    else
    {
        for (i = 0; i < n; i++)
            positive_factor[factor[i]] = i;
        positive_power = - power;
    }

    /*
     * Compose product[i] with positive_factor to the positive_power.
     */
    while (--positive_power >= 0)
        for (i = 0; i < n; i++)
            product[i] = positive_factor[product[i]];

    /*
     * Free scratch space.
     */
    my_free(positive_factor);
}

static void Zn_to_Sn(
    int element_of_Zn,
    int *element_of_Sn,
    int n)
{
    /*
     * Convert an abstract element of  $\mathbb{Z}/n$  to the corresponding
     * permutation in  $S(n)$ . Use the natural correspondence which maps
     * each element  $k$  of  $\mathbb{Z}/n$  to the permutation  $i \rightarrow i+k \pmod n$  in  $S(n)$ .
     */

    int i;

    for (i = 0; i < n; i++)
        element_of_Sn[i] = (i + element_of_Zn) % n;

    return;
}

static void compute_covering_type(
    RepresentationIntoSn *representation,
    int num_generators,
    int n)
{
    /*
     * Please see cover.h for the definition of a regular cover.
     * To check whether the given representation defines a regular cover,
     * it's sufficient to check whether for each sheet  $k$ , there is

```

```

    * a covering transformation taking sheet 0 to sheet k.
    *
    * While we're at it, we'll check whether the group of covering
    * transformations is cyclic of order n. A generator for a cyclic
    * group of covering transformations is a transformation whose
    * powers take sheet 0 to each of the other sheets.
    */

Boolean *transformation_found,
        progress;
int      k,
        *f,
        i,
        count,
        sheet,
        gen,
        nbr,
        nbr_must_map_to,
        image_of_zero,
        subgroup_order;

/*
 * Keep track of the values of k for which we have found a
 * covering transformation taking sheet 0 to sheet k.
 */
transformation_found = NEW_ARRAY(n, Boolean);
for (k = 0; k < n; k++)
    transformation_found[k] = FALSE;

/*
 * Allocate space for the permutation f() which a covering
 * transformation induces on the sheets.
 */
f = NEW_ARRAY(n, int);

/*
 * Attempt to find a covering transformation taking sheet 0
 * to each sheet k.
 */
for (k = 0; k < n; k++)
{
    /*
     * Skip values of k for which we've already found a covering
     * transformation (as a power of some previously computed
     * transformation).
     */
    if (transformation_found[k] == TRUE)
        continue;

    /*
     * Let f() be a permutation of the sheets {0, 1, ..., n-1}
     * which, we hope, will define a covering transformation.
     * Initialize f() to all -1's, to show that no values have
     * been decided yet.
     */
    for (i = 0; i < n; i++)
        f[i] = -1;

    /*
     * As an "unnecessary" error check, count how many values
     * of f() get assigned.
     */
    count = 0;

    /*
     * Set f(0) = k.
     */
    f[0] = k;
    count++;

    /*
     * Use the assignment of permutations to generators to deduce
     * the remaining values of f(). If we obtain contradictory
     * values for some f(sheet), then we know that there is no

```



```

    * covering transformation taking sheet 0 to sheet k.
    */
do
{
    progress = FALSE;

    for (sheet = 0; sheet < n; sheet++)
    {
        /*
        * If we don't yet know where the covering transformation
        * takes this sheet, then we can't deduce where it takes
        * its neighbors.
        */
        if (f[sheet] == -1)
            continue;

        /*
        * If we do know where the covering transformation takes
        * takes this sheet, then we can deduce where it must
        * take the neighbors. Consider each neighbor.
        */
        for (gen = 0; gen < num_generators; gen++)
        {
            nbr = representation->image[gen][sheet];
            nbr_must_map_to = representation->image[gen][f[sheet]];
            if (f[nbr] == -1)
            {
                f[nbr] = nbr_must_map_to;
                count++;
                progress = TRUE;
            }
            else
            {
                if (f[nbr] != nbr_must_map_to)
                {
                    representation->covering_type = irregular_cover;
                    my_free(transformation_found);
                    my_free(f);
                    return;
                }
            }
        }
    }
}

} while (progress == TRUE);

/*
* The covering is already known to be connected,
* so we should have assigned exactly n value to f().
*/
if (count != n)
    uFatalError("compute_covering_type", "representations");

/*
* The last time through the above loop we assigned no new
* values of f(), but it did serve to check that f() is
* consistent with all the gluings. In other words, f()
* is now known to be a valid covering transformation.
*/

/*
* We can save a little computation time by looking at the
* powers of f(), and seeing where they take sheet 0.
* While we're at it, compute the order of the subgroup
* of the group of covering transformations generated by f().
*/
image_of_zero = 0;
subgroup_order = 0;
do
{
    transformation_found[image_of_zero] = TRUE;
    subgroup_order++;
    image_of_zero = f[image_of_zero];
} while (image_of_zero != 0);

```

```

    /*
     * If f() generates the whole group of covering transformations,
     * then the cover is cyclic.
     */
    if (subgroup_order == n)
    {
        representation->covering_type = cyclic_cover;
        my_free(transformation_found);
        my_free(f);
        return;
    }
}

/*
 * If the covering were either irregular or cyclic, we would have
 * returned from within the above loop. So it must be regular.
 */
representation->covering_type = regular_cover;

/*
 * Free the local storage.
 */
my_free(transformation_found);
my_free(f);
}

void free_representation_list(
    RepresentationList *representation_list)
{
    RepresentationIntoSn *dead_representation;

    while (representation_list->list != NULL)
    {
        dead_representation = representation_list->list;
        representation_list->list = representation_list->list->next;
        free_representation(dead_representation, representation_list->num_generators,
            representation_list->num_cusps);
    }

    my_free(representation_list);
}

static void free_representation(
    RepresentationIntoSn *representation,
    int num_generators,
    int num_cusps)
{
    int i;

    for (i = 0; i < num_generators; i++)
        my_free(representation->image[i]);
    my_free(representation->image);

    for (i = 0; i < num_cusps; i++)
        my_free(representation->primitive_Deahn_image[i]);
    my_free(representation->primitive_Deahn_image);

    my_free(representation);
}

```