

```

/*
 * direct_product.c
 *
 * This file contains the function
 *
 * Boolean is_group_direct_product(SymmetryGroup *the_group);
 *
 * which checks whether the given group is a nonabelian, nontrivial direct
 * product. If so, it sets the_group's is_direct_product field to TRUE,
 * sets factor[0] and factor[1] to point to the factors, and calls
 * recognize_group() for each factor. factor[0] and factor[1] may
 * themselves be nonabelian, nontrivial direct products, leading to a
 * tree structure for the factorization.
 *
 * If the_group is not a nonabelian, nontrivial direct product, then
 * is_group_direct_product() sets the_group's is_direct_product field to
 * FALSE, and factor[0] and factor[1] to NULL.
 *
 * is_group_direct_product() assumes all of the_group's fields except
 * is_direct_product and factor[] have already been set.
 *
 * Overview of algorithm.
 *
 * The plan is to find all normal subgroups of the_group, then look
 * for a pair satisfying
 *
 * Theorem A. Let H and K be normal subgroups of a group G.
 * If H and K satisfy
 *
 * (1) The intersection of H and K contains only the identity.
 *
 * (2) For each h in H and k in K, hk = kh.
 *
 * (3)  $|G| = |H| |K|$ .
 *
 * then G is isomorphic to H x K.
 *
 * Proof. Define a map  $f: H \times K \rightarrow G$  by sending (h, k) to hk.
 *
 * Hypothesis (2) implies that f is a homomorphism.
 *  $f((h, k)(h', k')) = f((hh', kk')) = hh'kk' = hkh'k' =$ 
 *  $f((h, k)) f((h', k'))$ .
 *
 * Hypothesis (1) implies that f is one-to-one.  $f((h, k)) = id$ 
 *  $\Rightarrow hk = id \Rightarrow h = k^{-1} \Rightarrow h$  (resp. k) is an element of both
 * H and K, so therefore must be the identity  $\Rightarrow (h, k) = (id, id)$ .
 *
 * Hypothesis (3) implies that f is onto.  $|H \times K| = |H| |K| = |G|$ ,
 * so surjectivity follows from injectivity by the pidgeonhole
 * principle.
 *
 * Q.E.D.
 *
 * Terminology.
 *
 * Throughout this file, WE CONSIDER ONLY NORMAL SUBGROUPS. For example,
 * when we talk about the subgroup generated by an element, we mean the
 * smallest normal subgroup containing the element.
 *
 *  $H < K$  means the subgroup H is contained in the subgroup K.
 *
 *  $[g_1, g_2, \dots, g_n]$  means the subgroup generated by the elements
 *  $\{g_1, g_2, \dots, g_n\}$ .
 *
 * We define the "rank" of a subgroup to be the smallest number of elements
 * required to generate it. (As a special case, the subgroup {id} has
 * rank zero.) I'm not sure whether  $H < K$  implies  $rank(H) \leq rank(K)$ ,
 * so we make no assumptions on this matter.
 *
 * Finding all normal subgroups.

```

```

* Step 0. The trivial subgroup {id} has rank 0.
*
* Step 1. Find all subgroups generated by single elements. This gives
* all subgroups of rank <= 1.
*
* Step 2. Find all subgroups generated by the union of two rank 1
* subgroups. This gives all subgroups of rank <= 2.
*
* Step n. Find all subgroups generated by the union of a rank n subgroup and
* a rank 1 subgroup. This gives all subgroups of rank <= n + 1.
*
* The algorithm terminates when the only subgroup of some rank n is
* the group itself.
*
* Theorem B. In the above algorithm, a subgroup of rank n will appear at
* step n, but not sooner.
*
* Proof. First note that no rank n subgroup can appear before step n,
* because each subgroup appearing at step n is the smallest normal subgroup
* containing the generators of the n subgroups of rank 1 used to create it.
* To prove that every subgroup of rank n does indeed appear at step n,
* we induct on n. The theorem is clearly true for n = 0 and n = 1,
* so let H be a subgroup of rank n > 1. H is the smallest normal subgroup
* generated by some n-element set {h1, h2, ..., hn}. In symbols,
* H = [h1, h2, ..., hn]. Consider K = [h1, h2, ..., h(n-1)].
* K has rank exactly n-1 (clearly k has rank at most n-1, because it's
* generated by n-1 elements, and if it were generated by fewer than
* n-1 elements, say K = [k1, k2, ..., k(n-2)], then we'd have
* H = [k1, k2, ..., k(n-2), hn], contradicting the assumption that
* H has rank n). By induction, K appeared at step n-1. Therefore H will
* appear at step n, as the smallest normal subgroup containing the
* rank n-1 subgroup K and the rank 1 subgroup [hn].
* Note that the algorithm will not terminate while nontrivial
* subgroups remain unaccounted for, because the existence of a nontrivial
* subgroup of rank n implies the existence of a nontrivial subgroup
* of rank n-1 (as shown in the parenthetical remark in the preceding
* paragraph). Q.E.D.
*
* Comment. Even though subgroups of rank n first appear at step n,
* they may appear again at subsequent steps.
*/

```

```
#include "kernel.h"
```

```

typedef struct symmetry_subgroup
{
    /*
     * A subgroup of a group of order n is represented as a Boolean array
     * "contains" of length n. The element contains[i] will be TRUE iff
     * the subgroup contain element i.
     */
    Boolean                *contains;

    /*
     * How many elements does the whole group contain?
     * In other words, what's the length of the contains[] array?
     */
    int                    group_order;

    /*
     * How many elements does this subgroup contain?
     * In other words, how many of the elements in the contains[]
     * array are TRUE?
     */
    int                    subgroup_order;

    /*
     * We'll keep subgroups on NULL-terminated linked lists.
     */
    struct symmetry_subgroup *next;
} SymmetrySubgroup;

```

```

static Boolean      group_is_abelian_or_polyhedral(SymmetryGroup *the_group);
static SymmetrySubgroup **new_subgroup_array(int the_length);
static void         free_subgroup_array(SymmetrySubgroup **the_array, int the_length);
static SymmetrySubgroup *new_symmetry_subgroup(int order_of_group);
static void         free_symmetry_subgroup(SymmetrySubgroup *the_subgroup);
static void         compute_rank_zero_subgroup(SymmetryGroup *the_group,
SymmetrySubgroup **the_list);
static void         compute_rank_one_subgroups(SymmetryGroup *the_group,
SymmetrySubgroup **the_list);
static void         compute_rank_n_plus_one_subgroups(SymmetryGroup *the_group,
SymmetrySubgroup **subgroup_of_rank, int n);
static void         find_subgroup_generated(SymmetryGroup *the_group, SymmetrySubgroup *
the_subset);
static void         add_conjugates(SymmetryGroup *the_group, SymmetrySubgroup *
the_subset);
static void         add_products(SymmetryGroup *the_group, SymmetrySubgroup *
the_subset);
static Boolean      subgroup_on_some_list(SymmetrySubgroup *the_subgroup,
SymmetrySubgroup **subgroup_of_rank);
static Boolean      subgroup_on_list(SymmetrySubgroup *the_subgroup, SymmetrySubgroup *
the_list);
static Boolean      same_subgroup(SymmetrySubgroup *subgroupA, SymmetrySubgroup *
subgroupB);
static Boolean      is_subset(SymmetrySubgroup *subgroupA, SymmetrySubgroup *subgroupB);
static SymmetrySubgroup *find_union(SymmetrySubgroup *subgroupA, SymmetrySubgroup *
subgroupB);
static void         add_subgroup_to_list(SymmetrySubgroup *the_subgroup,
SymmetrySubgroup **the_list);
static void         sort_by_order(int array_length, SymmetrySubgroup **subgroup_of_rank,
SymmetrySubgroup **subgroup_of_order);
static Boolean      search_for_direct_product(SymmetryGroup *the_group,
SymmetrySubgroup **subgroup_of_order);
static Boolean      theoremA(SymmetryGroup *the_group, SymmetrySubgroup *subgroup_H,
SymmetrySubgroup *subgroup_K);
static Boolean      condition1(SymmetryGroup *the_group, SymmetrySubgroup *subgroup_H,
SymmetrySubgroup *subgroup_K);
static Boolean      condition2(SymmetryGroup *the_group, SymmetrySubgroup *subgroup_H,
SymmetrySubgroup *subgroup_K);
static Boolean      condition3(SymmetryGroup *the_group, SymmetrySubgroup *subgroup_H,
SymmetrySubgroup *subgroup_K);
static SymmetryGroup *subgroup_to_group(SymmetryGroup *the_whole_group, SymmetrySubgroup *
the_subgroup);
static void         set_up_index_conversion(SymmetrySubgroup *the_subgroup, int **
new_to_old_indices, int **old_to_new_indices);
static void         copy_symmetries(SymmetryGroup *the_whole_group, SymmetryGroup *
the_subgroup, int new_to_old_indices[]);
static void         copy_one_symmetry(Symmetry *source, Symmetry **dest);
static void         copy_multiplication_table(SymmetryGroup *the_whole_group,
SymmetryGroup *the_subgroup, int new_to_old_indices[], int old_to_new_indices[]);
static void         copy_element_orders(SymmetryGroup *the_whole_group, SymmetryGroup *
the_subgroup, int new_to_old_indices[]);
static void         copy_inverses(SymmetryGroup *the_whole_group, SymmetryGroup *
the_subgroup, int new_to_old_indices[], int old_to_new_indices[]);

```

```

Boolean is_group_direct_product(
    SymmetryGroup *the_group)
{
    SymmetrySubgroup **subgroup_of_rank,
    **subgroup_of_order;
    int n;
    Boolean success;

    /*
     * If the group is known to be abelian or polyhedral,
     * then it's not a nonabelian, nontrivial direct product.
     */
    if (group_is_abelian_or_polyhedral(the_group) == TRUE)
        return FALSE;

    /*
     * During the first half of the algorithm, the SymmetrySubgroups
     * will be organized by rank. subgroup_of_rank[] will be an array

```

```

    * of pointers of length (the_group->order + 1). subgroup_of_rank[n]
    * will point to the first element in the NULL-terminated linked list
    * of SymmetrySubgroups of rank n. Obviously all but the first
    * few elements in the subgroup_of_rank[] array will be NULL, since
    * there won't be any subgroups of very high rank.
    *
    * During the second half of the algorithm, a similar system will
    * organize the subgroups by order (i.e. number of elements) on the
    * array subgroup_of_order[]. subgroup_of_order[n] will point to the
    * first element in the NULL-terminated linked list of SymmetrySubgroups
    * of order n. For most values of n there won't be any subgroups of
    * that order, so most elements of subgroup_of_order[] will be NULL.
    */
subgroup_of_rank = new_subgroup_array(the_group->order + 1);
subgroup_of_order = new_subgroup_array(the_group->order + 1);

/*
 * The present algorithm has no use for the rank 0 subgroup {id},
 * but we compute it anyhow just in case this code is ever needed
 * for any other purpose.
 */
compute_rank_zero_subgroup(the_group, &subgroup_of_rank[0]);

/*
 * Compute all rank 1 subgroups.
 */
compute_rank_one_subgroups(the_group, &subgroup_of_rank[1]);

/*
 * Find all subgroups generated by the union of a rank n subgroup and a
 * rank 1 subgroup, and store them on the list of rank n+1 subgroups.
 * Keep going as long as we keep finding new subgroups.
 */
for (n = 1; subgroup_of_rank[n] != NULL; n++)
    compute_rank_n_plus_one_subgroups(the_group, subgroup_of_rank, n);

/*
 * Sort the subgroups by their orders. We no longer care about
 * the ranks.
 */
sort_by_order(the_group->order + 1, subgroup_of_rank, subgroup_of_order);

/*
 * Try to find a pair of subgroups satisfying Theorem A.
 */
success = search_for_direct_product(the_group, subgroup_of_order);

free_subgroup_array(subgroup_of_rank, the_group->order + 1);
free_subgroup_array(subgroup_of_order, the_group->order + 1);

return success;
}

static Boolean group_is_abelian_or_polyhedral(
    SymmetryGroup *the_group)
{
    if (the_group->is_abelian == TRUE
        || the_group->is_polyhedral == TRUE)
    {
        the_group->is_direct_product = FALSE;

        the_group->factor[0] = NULL;
        the_group->factor[1] = NULL;

        return TRUE;
    }
    else
        return FALSE;
}

static SymmetrySubgroup **new_subgroup_array(
    int the_length)

```

```

{
    SymmetrySubgroup    **the_array;
    int                 i;

    the_array = NEW_ARRAY(the_length, SymmetrySubgroup *);

    for (i = 0; i < the_length; i++)
        the_array[i] = NULL;

    return the_array;
}

static void free_subgroup_array(
    SymmetrySubgroup    **the_array,
    int                 the_length)
{
    int                 i;
    SymmetrySubgroup    *dead_subgroup;

    for (i = 0; i < the_length; i++)
        while(the_array[i] != NULL)
        {
            dead_subgroup = the_array[i];
            the_array[i] = the_array[i]->next;
            free_symmetry_subgroup(dead_subgroup);
        }

    my_free(the_array);
}

static SymmetrySubgroup *new_symmetry_subgroup(
    int                 order_of_group)
{
    /*
     * Allocate a new SymmetrySubgroup and initialize it to the empty
     * subgroup. order_of_group is the order of the whole group,
     * not the subgroup.
     */

    SymmetrySubgroup    *new_subgroup;
    int                 i;

    new_subgroup = NEW_STRUCT(SymmetrySubgroup);
    new_subgroup->contains = NEW_ARRAY(order_of_group, Boolean);
    new_subgroup->group_order = order_of_group;
    new_subgroup->subgroup_order = 0;
    new_subgroup->next = NULL;

    for (i = 0; i < new_subgroup->group_order; i++)
        new_subgroup->contains[i] = FALSE;

    return new_subgroup;
}

static void free_symmetry_subgroup(
    SymmetrySubgroup    *the_subgroup)
{
    my_free(the_subgroup->contains);
    my_free(the_subgroup);
}

static void compute_rank_zero_subgroup(
    SymmetryGroup        *the_group,
    SymmetrySubgroup    **the_list)
{
    /*
     * There's exactly one rank 0 subgroup, namely {id}.
     */

```

```

/*
 * First an error check.
 */

if (*the_list != NULL)
    uFatalError("compute_rank_zero_subgroup", "direct_product");

/*
 * Allocate the SymmetrySubgroup and initialize it
 * to the empty subgroup.
 */

(*the_list) = new_symmetry_subgroup(the_group->order);

/*
 * Add the identity element.
 */

(*the_list)->contains[0] = TRUE;
(*the_list)->subgroup_order = 1;
(*the_list)->next = NULL;
}

static void compute_rank_one_subgroups(
    SymmetryGroup      *the_group,
    SymmetrySubgroup   **the_list)
{
    SymmetrySubgroup   *the_subgroup;
    int                 g;

    /*
     * First an error check.
     */

    if (*the_list != NULL)
        uFatalError("compute_rank_one_subgroups", "direct_product");

    /*
     * For each element of the_group we compute the smallest normal
     * subgroup containing it. If the subgroup is not already on the_list,
     * we add it. We do not consider the subgroup containing only the
     * identity, because we want only rank 1 subgroups, not rank 0.
     */

    for (g = 1; g < the_group->order; g++)
    {
        /*
         * Allocate and initialize the_subgroup.
         */
        the_subgroup = new_symmetry_subgroup(the_group->order);

        /*
         * Add the identity and the element g.
         */
        the_subgroup->contains[0] = TRUE;
        the_subgroup->contains[g] = TRUE;
        the_subgroup->subgroup_order = 2;

        /*
         * Compute the smallest normal subgroup containing g.
         */
        find_subgroup_generated(the_group, the_subgroup);

        /*
         * Add it to the_list if it's not already there.
         * Otherwise destroy it.
         */
        if (subgroup_on_list(the_subgroup, *the_list) == FALSE)
            add_subgroup_to_list(the_subgroup, the_list);
        else
            free_symmetry_subgroup(the_subgroup);
    }
}

```

```

static void compute_rank_n_plus_one_subgroups(
    SymmetryGroup      *the_group,
    SymmetrySubgroup   **subgroup_of_rank,
    int                n)
{
    SymmetrySubgroup   *rank_n_subgroup,
                      *rank_one_subgroup,
                      *the_union;

    /*
     * Consider all rank n subgroups.
     */
    for (    rank_n_subgroup = subgroup_of_rank[n];
          rank_n_subgroup != NULL;
          rank_n_subgroup = rank_n_subgroup->next)

        /*
         * Consider all rank one subgroups.
         */
        for (    rank_one_subgroup = subgroup_of_rank[1];
              rank_one_subgroup != NULL;
              rank_one_subgroup = rank_one_subgroup->next)
        {
            /*
             * If one of {rank_n_subgroup, rank_one_subgroup} is
             * contained in the other, move on.
             */
            if (is_subset(rank_one_subgroup, rank_n_subgroup) == TRUE
                || is_subset(rank_n_subgroup, rank_one_subgroup) == TRUE)
                continue;

            /*
             * Find the union of the rank_n_subgroup and the
             * rank_one_subgroup . . .
             */
            the_union = find_union(rank_n_subgroup, rank_one_subgroup);

            /*
             * . . . and then find the smallest normal subgroup
             * which contains it.
             */
            find_subgroup_generated(the_group, the_union);

            /*
             * Add it to the list of rank (n+1) subgroups iff it hasn't
             * already been computed as a subgroup of rank (n+1) or less.
             * Otherwise destroy it.
             */
            if (subgroup_on_some_list(the_union, subgroup_of_rank) == FALSE)
                add_subgroup_to_list(the_union, &subgroup_of_rank[n+1]);
            else
                free_symmetry_subgroup(the_union);
        }
    }
}

static void find_subgroup_generated(
    SymmetryGroup      *the_group,
    SymmetrySubgroup   *the_subset)
{
    /*
     * This routine takes a subset of a group and finds the smallest normal
     * subgroup containing it. The result is computed "in place", that is,
     * the original subset is replaced by the normal subgroup it generates.
     *
     *
     * We want to insure that the_subset is closed under multiplication
     * (so it's a subgroup) and closed under conjugation (so it's normal).
     *
     * Lemma. It suffices to first take the closure under conjugation,
     * and then take the closure under multiplication. In other words,
     * taking the closure under multiplication won't destroy the closure
    */
}

```

```

    * under conjugacy.
    *
    * Proof. After first taking the closure of the_subset under conjugation,
    * we get a set of elements of the form {caC, dbD, ...}. Taking the
    * closure under multiplication gives all possible products of these
    * elements: a typical element has the form (caC)...(dbD). If we
    * conjugate such an element by any other element e in the group we
    * get something of the form e((caC)...(dbD))E = (ecaCE)...(edbDE)
    * = (ec)a(CE)...(ed)b(DE) which is again a product of conjugates of
    * elements of the original subset. Q.E.D.
    */

    add_conjugates(the_group, the_subset);
    add_products(the_group, the_subset);
}

static void add_conjugates(
    SymmetryGroup      *the_group,
    SymmetrySubgroup   *the_subset)
{
    int    g,
           h,
           ghg;

    /*
     * Add all conjugates of all elements of the_subset.
     * This algorithm is a little simpler than the one in add_products().
     * Here we needn't worry about conjugates of conjugates
     * (because  $b(ax(a^{-1}))(b^{-1}) = (ba)x(ba)^{-1}$ ), whereas in
     * add_products() we did need to worry about products of products.
     */

    /*
     * Begin with a quick error check.
     */
    if (the_group->order != the_subset->group_order)
        uFatalError("add_conjugates", "direct_product");

    /*
     * For each element h in the_subset . . .
     */
    for (h = 0; h < the_subset->group_order; h++)
    {
        if (the_subset->contains[h] == FALSE)
            continue;

        /*
         * . . . and each element g in the whole group . . .
         */
        for (g = 0; g < the_group->order; g++)
        {
            /*
             * . . . compute gh(g^{-1}).
             */
            ghg = the_group->product
                [the_group->product[g][h]]
                [the_group->inverse[g]];

            /*
             * If gh(g^{-1}) isn't already in the subset, add it.
             */
            if (the_subset->contains[ghg] == FALSE)
            {
                the_subset->contains[ghg] = TRUE;
                the_subset->subgroup_order++;
            }
        }
    }
}

static void add_products(
    SymmetryGroup      *the_group,

```



```

SymmetrySubgroup    *the_subset)
{
    int    *the_queue,
           the_product,
           i,
           j,
           count;

    /*
     * We want to insure that the final subset contains the products
     * of all pairs of its elements, not merely pairs of elements of
     * the original subset.  Furthermore, don't want to multiply any
     * particular pair of elements more than one.  The way to do this
     * is to put the elements of the_subset onto a queue.  We'll work
     * our way along the queue, checking the product of each successive
     * element with all preceding elements (as well as with itself).
     * As new elements are found, they are added to the end of the queue.
     */

    /*
     * Begin with a quick error check.
     */
    if (the_group->order != the_subset->group_order)
        uFatalError("add_products", "direct_product");

    /*
     * Allocate space for the_queue.
     */
    the_queue = NEW_ARRAY(the_group->order, int);

    /*
     * Copy the original elements of the_subset onto the_queue.
     */
    for (i = 0, count = 0; i < the_group->order; i++)
        if (the_subset->contains[i] == TRUE)
            the_queue[count++] = i;

    /*
     * Make sure we found the number of elements we expected.
     */
    if (count != the_subset->subgroup_order)
        uFatalError("add_products", "direct_product");

    /*
     * For each element on the_queue, compute the product with all
     * preceding elements (and with the element itself), and see whether
     * any new elements are generated.
     *
     * Technical note:  the_subset->subgroup_order may increase as we
     * go through the loop, but will never exceed the_subset->group_order.
     */

    for (i = 0; i < the_subset->subgroup_order; i++)
        for (j = 0; j <= i; j++)
        {
            the_product = the_group->product[the_queue[i]][the_queue[j]];

            if (the_subset->contains[the_product] == FALSE)
            {
                the_subset->contains[the_product] = TRUE;
                the_queue[the_subset->subgroup_order++] = the_product;
            }

            the_product = the_group->product[the_queue[j]][the_queue[i]];

            if (the_subset->contains[the_product] == FALSE)
            {
                the_subset->contains[the_product] = TRUE;
                the_queue[the_subset->subgroup_order++] = the_product;
            }
        }

    /*

```

```

    * Yet another error check.
    * The subgroup_order should not exceed the group_order, and
    * the subgroup_order should divide the group_order.
    */
    if (the_subset->subgroup_order > the_subset->group_order
        || the_subset->group_order % the_subset->subgroup_order != 0)
        uFatalError("add_products", "direct_product");

    /*
    * Free the_queue.
    */
    my_free(the_queue);
}

static Boolean subgroup_on_some_list(
    SymmetrySubgroup *the_subgroup,
    SymmetrySubgroup **subgroup_of_rank)
{
    int i;

    for (i = 0; i <= the_subgroup->group_order; i++)
    {
        /*
        * If there are no rank i subgroups, there won't be any of
        * rank greater than i either.
        */
        if (subgroup_of_rank[i] == NULL)
            return FALSE;

        /*
        * Check whether the_subgroup is already on the rank i list.
        */
        if (subgroup_on_list(the_subgroup, subgroup_of_rank[i]) == TRUE)
            return TRUE;
    }

    /*
    * We should always return from within the above loop,
    * because there can't possibly be a subgroup whose rank equals
    * the rank of the group. (The identity doesn't count as a generator,
    * so in a group of order n it's a priori impossible to have more
    * than (n-1) generators.)
    */
    uFatalError("subgroup_on_some_list", "direct_product");

    /*
    * The C++ compiler would like a return value, even though
    * we never return from the uFatalError() call.
    */
    return TRUE;
}

static Boolean subgroup_on_list(
    SymmetrySubgroup *the_subgroup,
    SymmetrySubgroup *the_list)
{
    SymmetrySubgroup *a_subgroup;

    for (a_subgroup = the_list;
         a_subgroup != NULL;
         a_subgroup = a_subgroup->next)

        if (same_subgroup(the_subgroup, a_subgroup) == TRUE)
            return TRUE;

    return FALSE;
}

static Boolean same_subgroup(
    SymmetrySubgroup *subgroupA,
    SymmetrySubgroup *subgroupB)

```

```

{
    int i;

    if (subgroupA->group_order != subgroupB->group_order)
        uFatalError("same_subgroup", "direct_product");

    for (i = 0; i < subgroupA->group_order; i++)
        if (subgroupA->contains[i] != subgroupB->contains[i])
            return FALSE;

    return TRUE;
}

static Boolean is_subset(
    SymmetrySubgroup *subgroupA,
    SymmetrySubgroup *subgroupB)
{
    int i;

    if (subgroupA->group_order != subgroupB->group_order)
        uFatalError("is_subset", "direct_product");

    for (i = 0; i < subgroupA->group_order; i++)
        if (subgroupA->contains[i] == TRUE
            && subgroupB->contains[i] == FALSE)
            return FALSE;

    return TRUE;
}

static SymmetrySubgroup *find_union(
    SymmetrySubgroup *subgroupA,
    SymmetrySubgroup *subgroupB)
{
    SymmetrySubgroup *the_union;
    int i;

    if (subgroupA->group_order != subgroupB->group_order)
        uFatalError("find_union", "direct_product");

    the_union = new_symmetry_subgroup(subgroupA->group_order);

    for (i = 0; i < the_union->group_order; i++)
    {
        the_union->contains[i] =
            subgroupA->contains[i] || subgroupB->contains[i];
        if (the_union->contains[i])
            the_union->subgroup_order++;
    }

    return the_union;
}

static void add_subgroup_to_list(
    SymmetrySubgroup *the_subgroup,
    SymmetrySubgroup **the_list)
{
    the_subgroup->next = *the_list;
    *the_list = the_subgroup;
}

static void sort_by_order(
    int array_length,
    SymmetrySubgroup **subgroup_of_rank,
    SymmetrySubgroup **subgroup_of_order)
{
    int i;
    SymmetrySubgroup *the_subgroup;

    for (i = 0; i < array_length; i++)

```

```

    while (subgroup_of_rank[i] != NULL)
    {
        the_subgroup      = subgroup_of_rank[i];
        subgroup_of_rank[i] = subgroup_of_rank[i]->next;
        add_subgroup_to_list(
            the_subgroup,
            &subgroup_of_order[the_subgroup->subgroup_order]);
    }
}

static Boolean search_for_direct_product(
    SymmetryGroup      *the_group,
    SymmetrySubgroup   **subgroup_of_order)
{
    /*
     * Look for subgroups H and K satisfying Theorem A
     * of the documentation at the top of this file.
     *
     * Let  $n = |H|$  and  $m = |K|$ . Without loss of generality, assume  $n \leq m$ .
     *
     * We need consider only those values of  $n$  satisfying  $n^2 \leq |G|$ .
     */

    int          n,
                m;
    SymmetrySubgroup *subgroup_H,
                *subgroup_K;

    for (n = 2; n*n <= the_group->order; n++)
    {
        /*
         * By LaGrange's Theorem,  $n$  must divide the order of the group.
         */
        if (the_group->order % n != 0)
            continue;

        m = the_group->order / n;

        /*
         * Consider all normal subgroups H . . .
         */
        for (subgroup_H = subgroup_of_order[n];
             subgroup_H != NULL;
             subgroup_H = subgroup_H->next)

            /*
             * . . . and K.
             */
            for (subgroup_K = subgroup_of_order[m];
                 subgroup_K != NULL;
                 subgroup_K = subgroup_K->next)

                if (theoremA(the_group, subgroup_H, subgroup_K) == TRUE)
                {
                    the_group->is_direct_product = TRUE;

                    the_group->factor[0] = subgroup_to_group(the_group, subgroup_H);
                    the_group->factor[1] = subgroup_to_group(the_group, subgroup_K);

                    return TRUE;
                }
    }

    /*
     * No luck.
     */

    the_group->is_direct_product = FALSE;

    the_group->factor[0] = NULL;
    the_group->factor[1] = NULL;
}

```

```

    return FALSE;
}

static Boolean theoremA(
    SymmetryGroup      *the_group,
    SymmetrySubgroup    *subgroup_H,
    SymmetrySubgroup    *subgroup_K)
{
    if (subgroup_H->group_order != subgroup_K->group_order)
        uFatalError("theoremA", "direct_product");

    return(
        condition1(the_group, subgroup_H, subgroup_K)
        && condition2(the_group, subgroup_H, subgroup_K)
        && condition3(the_group, subgroup_H, subgroup_K));
}

static Boolean condition1(
    SymmetryGroup      *the_group,
    SymmetrySubgroup    *subgroup_H,
    SymmetrySubgroup    *subgroup_K)
{
    /*
     * Does the intersection of H and K contain only the identity?
     */

    int g;

    /*
     * First make sure both H and K do in fact contain the identity.
     */

    if (subgroup_H->contains[0] == FALSE
        || subgroup_K->contains[0] == FALSE)
        uFatalError("condition1", "direct_product");

    /*
     * Now check that they have no other elements in common.
     */

    for (g = 1; g < the_group->order; g++)

        if (subgroup_H->contains[g] == TRUE
            && subgroup_K->contains[g] == TRUE)

            return FALSE;

    return TRUE;
}

static Boolean condition2(
    SymmetryGroup      *the_group,
    SymmetrySubgroup    *subgroup_H,
    SymmetrySubgroup    *subgroup_K)
{
    /*
     * Check that for each h in H and k in K, hk = kh.
     */

    int h,
        k;

    for (h = 0; h < subgroup_H->group_order; h++)
    {
        if (subgroup_H->contains[h] == FALSE)
            continue;

        for (k = 0; k < subgroup_K->group_order; k++)
        {
            if (subgroup_K->contains[k] == FALSE)
                continue;

```

```

        if (the_group->product[h][k] != the_group->product[k][h])
            return FALSE;
    }

}

return TRUE;
}

static Boolean condition3(
    SymmetryGroup      *the_group,
    SymmetrySubgroup   *subgroup_H,
    SymmetrySubgroup   *subgroup_K)
{
    /*
     * Condition 3 should already have been implicitly verified
     * in search_for_direct_product().
     */

    if (subgroup_H->subgroup_order * subgroup_K->subgroup_order
        != the_group->order)

        uFatalError("condition3", "direct_product");

    return TRUE;
}

static SymmetryGroup *subgroup_to_group(
    SymmetryGroup      *the_whole_group,
    SymmetrySubgroup   *the_subgroup)
{
    SymmetryGroup      *the_group;
    int                 *new_to_old_indices,
                        *old_to_new_indices;

    /*
     * subgroup_to_group() converts a SymmetrySubgroup
     * to a free-standing SymmetryGroup.
     */

    /*
     * Allocate the SymmetryGroup.
     */
    the_group = NEW_STRUCT(SymmetryGroup);

    /*
     * Set the_group's order.
     */
    the_group->order = the_subgroup->subgroup_order;

    /*
     * Allocate temporary arrays which will give the old element indices
     * (i.e. in the_whole_group) in terms of the new indices (i.e. in
     * the_group), and vice versa.
     */
    set_up_index_conversion(    the_subgroup,
                               &new_to_old_indices,
                               &old_to_new_indices);

    /*
     * Copy the appropriate Symmetries from the_whole_group to the_group
     */
    copy_symmetries(    the_whole_group,
                        the_group,
                        new_to_old_indices);

    /*
     * Copy the appropriate element in the multiplication table.
     */
    copy_multiplication_table( the_whole_group,
                              the_group,

```

```

        new_to_old_indices,
        old_to_new_indices);

/*
 * Copy the appropriate element orders.
 */
copy_element_orders(the_whole_group,
                    the_group,
                    new_to_old_indices);

/*
 * Copy the appropriate inverses.
 */
copy_inverses(      the_whole_group,
                    the_group,
                    new_to_old_indices,
                    old_to_new_indices);

/*
 * Attempt to recognize the_group.
 */
recognize_group(the_group);

/*
 * Free the temporary arrays.
 */
my_free(new_to_old_indices);
my_free(old_to_new_indices);

return the_group;
}

static void set_up_index_conversion(
    SymmetrySubgroup *the_subgroup,
    int **new_to_old_indices,
    int **old_to_new_indices)
{
    int old_index,
        new_index;

    *new_to_old_indices = NEW_ARRAY(the_subgroup->subgroup_order, int);
    *old_to_new_indices = NEW_ARRAY(the_subgroup->group_order, int);

    for (    old_index = 0, new_index = 0;
          old_index < the_subgroup->group_order;
          old_index++)

        if (the_subgroup->contains[old_index] == TRUE)
        {
            (*new_to_old_indices)[new_index] = old_index;
            (*old_to_new_indices)[old_index] = new_index;
            new_index++;
        }

    if (new_index != the_subgroup->subgroup_order)
        uFatalError("set_up_index_conversion", "direct_product");
}

static void copy_symmetries(
    SymmetryGroup *the_whole_group,
    SymmetryGroup *the_subgroup,
    int new_to_old_indices[])
{
    int i;

    /*
     * 96/11/30 Allow for the possibility that the_whole_group
     * has no SymmetryList.
     */
    if (the_whole_group->symmetry_list == NULL)
    {
        the_subgroup->symmetry_list = NULL;
    }
}

```

```

    return;
}

the_subgroup->symmetry_list = NEW_STRUCT(SymmetryList);

the_subgroup->symmetry_list->num_isometries = the_subgroup->order;
the_subgroup->symmetry_list->isometry = NEW_ARRAY( the_subgroup->order,
                                                    Isometry *);

for (i = 0; i < the_subgroup->order; i++)

    copy_one_symmetry(
        the_whole_group->symmetry_list->isometry[new_to_old_indices[i]],
        &the_subgroup->symmetry_list->isometry[i]);
}

static void copy_one_symmetry(
    Symmetry *source,
    Symmetry **dest)
{
    int i,
        j,
        k;

    *dest = NEW_STRUCT(Isometry);

    (*dest)->num_tetrahedra = source->num_tetrahedra;
    (*dest)->num_cusps = source->num_cusps;

    (*dest)->tet_image = NEW_ARRAY(source->num_tetrahedra, int);
    (*dest)->tet_map = NEW_ARRAY(source->num_tetrahedra, Permutation);

    for (i = 0; i < source->num_tetrahedra; i++)
    {
        (*dest)->tet_image[i] = source->tet_image[i];
        (*dest)->tet_map[i] = source->tet_map[i];
    }

    (*dest)->cuspid_image = NEW_ARRAY(source->num_cusps, int);
    (*dest)->cuspid_map = NEW_ARRAY(source->num_cusps, MatrixInt22);

    for (i = 0; i < source->num_cusps; i++)
    {
        (*dest)->cuspid_image[i] = source->cuspid_image[i];
        for (j = 0; j < 2; j++)
            for (k = 0; k < 2; k++)
                (*dest)->cuspid_map[i][j][k] = source->cuspid_map[i][j][k];
    }

    (*dest)->extends_to_link = source->extends_to_link;
    (*dest)->next = NULL;
}

static void copy_multiplication_table(
    SymmetryGroup *the_whole_group,
    SymmetryGroup *the_subgroup,
    int new_to_old_indices[],
    int old_to_new_indices[])
{
    int i,
        j;

    /*
     * Allocate space for the multiplication table.
     */

    the_subgroup->product = NEW_ARRAY(the_subgroup->order, int *);
    for (i = 0; i < the_subgroup->order; i++)
        the_subgroup->product[i] = NEW_ARRAY(the_subgroup->order, int);

    /*
     * Fill in the entries.

```



```
    */
    for (i = 0; i < the_subgroup->order; i++)
        for (j = 0; j < the_subgroup->order; j++)
            the_subgroup->product[i][j] = old_to_new_indices[
                the_whole_group->product[new_to_old_indices[i]]
                    [new_to_old_indices[j]]];
}

static void copy_element_orders(
    SymmetryGroup      *the_whole_group,
    SymmetryGroup      *the_subgroup,
    int                new_to_old_indices[])
{
    int i;

    /*
     * Allocate the array which will hold the orders of the elements.
     */
    the_subgroup->order_of_element = NEW_ARRAY(the_subgroup->order, int);

    /*
     * Copy in the elements.
     */
    for (i = 0; i < the_subgroup->order; i++)
        the_subgroup->order_of_element[i] = the_whole_group->
            order_of_element[new_to_old_indices[i]];
}

static void copy_inverses(
    SymmetryGroup      *the_whole_group,
    SymmetryGroup      *the_subgroup,
    int                new_to_old_indices[],
    int                old_to_new_indices[])
{
    int i;

    /*
     * Allocate the array which will hold the inverses of the elements.
     */
    the_subgroup->inverse = NEW_ARRAY(the_subgroup->order, int);

    /*
     * Copy in the elements.
     */
    for (i = 0; i < the_subgroup->order; i++)
        the_subgroup->inverse[i] = old_to_new_indices[
            the_whole_group->inverse[new_to_old_indices[i]]];
}
```