

```

/*
 * elements_generate_group.c
 *
 * This file provides the utility
 *
 * Boolean elements_generate_group(
 *     SymmetryGroup *the_group,
 *     int           num_possible_generators,
 *     int           possible_generators[]);
 *
 * which the SymmetryGroup recognition functions use to check whether a
 * map from a well-known group to the_group is surjective.
 *
 * The array possible_generators[] contains num_possible_generators
 * elements of the_group. If those elements generate the_group, then
 * elements_generate_group() returns TRUE. Otherwise it returns FALSE.
 */

#include "kernel.h"

Boolean elements_generate_group(
    SymmetryGroup *the_group,
    int           num_possible_generators,
    int           possible_generators[])
{
    /*
     * Do the elements listed in the array possible_generators[]
     * generate the_group?
     *
     * We use a queue to keep track of which elements are in the
     * subgroup generated by possible_generators[]. As new elements
     * are discovered, they go on the back of the queue. We also
     * keep track of an initial segment of the queue containing those
     * elements which are "fully processed". An element has been fully
     * processed once we've considered its product with all other fully
     * processed elements (and with itself). If any such products
     * yield elements not yet on the queue, we add them to the queue.
     * Once all the elements on the queue are fully processed, we'll
     * have the smallest subgroup containing the possible_generators[].
     * (Proof: Clearly we'll have the smallest subset which is closed
     * under multiplication. For finite groups, closure under
     * multiplication implies closure under inverses -- the subgroup
     * generated by a single element is finite cyclic, so an element's
     * inverse is one of its powers.)
     *
     * The variable total_elements keeps track of the total number of
     * elements on the queue, while fully_processed keeps track of the
     * number which have been fully processed. For example, at some point
     * in the computation, the queue might look like
     *
     *      0  7  2  12  5  6  14  -1  -1  -1  -1  -1  -1  -1  -1  -1
     *
     * total_elements will equal 7, indicating that so far we've discovered
     * 7 elements of the subgroup generated by possible_generators[].
     *
     * fully_processed might equal, say, 4, indicating that all possible
     * products of the elements {0, 7, 2, 12} are among the 7 elements on
     * the queue.
     *
     * Naively this would be a cubic time algorithm, but we can do it in
     * quadratic time if we maintain a Boolean array recording which
     * elements are on the queue. That is, array[i] == TRUE iff element i
     * is on the queue.
     */

    int     *queue,
            total_elements,
            fully_processed;
    Boolean *array;
    int     g,
            product;
    int     i;

```

```

/*
 * Allocate space for the queue and the array.
 */
queue = NEW_ARRAY(the_group->order, int);
array = NEW_ARRAY(the_group->order, Boolean);

/*
 * Initialize the queue and the array.
 */
for (i = 0; i < the_group->order; i++)
{
    queue[i] = -1;
    array[i] = FALSE;
}

/*
 * Add the possible_generators[] to the queue.
 */
for (i = 0; i < num_possible_generators; i++)
{
    queue[i] = possible_generators[i];
    array[possible_generators[i]] = TRUE;
}

/*
 * Initialize the counts.
 */
total_elements = num_possible_generators;
fully_processed = 0;

/*
 * Process the elements on the queue.
 */
while (fully_processed < total_elements)
{
    /*
     * For convenience, call the next element on the queue "g".
     * Increment fully_processed, so that in the next step g will
     * be multiplied by itself as well as all preceding elements.
     */
    g = queue[fully_processed++];

    /*
     * Consider the products (on the left and on the right) of
     * g with all fully processed elements (including g itself).
     */
    for (i = 0; i < fully_processed; i++)
    {
        /*
         * If g * queue[i] isn't already on the queue, add it.
         */
        product = the_group->product[g][queue[i]];
        if (array[product] == FALSE)
        {
            queue[total_elements++] = product;
            array[product] = TRUE;
        }

        /*
         * If queue[i] * g isn't already on the queue, add it.
         */
        product = the_group->product[queue[i]][g];
        if (array[product] == FALSE)
        {
            queue[total_elements++] = product;
            array[product] = TRUE;
        }
    }
}

/*
 * Free local storage.
 */
my_free(queue);

```

```
my_free(array);

/*
 * possible_generators[] generate the whole group iff all the
 * group elements ended up on the queue.
 */
return (total_elements == the_group->order);
}
```