Minimum Generating Set Algorithm

Notes :

1. If the function "returns" at any point, it doesn't execute any more statements.
2. By "for any" , we mean : iterate over all possibilities one by one.
3. For a set $\mathbf{g}$, the group generated by using it as generating set is denoted by $< g >$

Algorithm

def $\text{minGen}(\mathbf{G})$:

First, we'll cover two base cases

First, we check if G is a cyclic group, in which case, we only need to return the cyclic generator G_1

if $\mathbf{G}$ is a cyclic group :

for any element $G_1 \in \mathbf{G}$:

if $< G_1 > = \mathbf{G}$:

return $\{G_1\}$

Next, If G is a simple group, then it can be generated by only 2 elements, say G_1, G_2

if $\mathbf{G}$ is a simple group :

for any elements $G_1, G_2 \in \mathbf{G}$:

if $< G_1, G_2 > = \mathbf{G}$:

return $\{G_1, G_2\}$

Now we move to the recursive part of the algorithm

Find any minimal normal subgroup $\mathbf{N}$ of $\mathbf{G}$

Find any generating set $\mathbf{n} = \{n_1, n_2 \dots n_k\}$ of $\mathbf{N}$

Let $\{g_1\mathbf{N}, g_2\mathbf{N} \dots g_l\mathbf{N}\} = \text{minGen}(\ \mathbf{G}/\mathbf{N}\ )$ for some $g_1, g_2 \dots g_l \in \mathbf{G}$

Let $\mathbf{g} = \{g_1, g_2, g_3 \dots g_l\}$

Essentially, $\mathbf{g}$ is the set of representative elements of the co-sets of N in the quotient group G/N

We'll modify $\mathbf{g}$ in various ways to get the set $\mathbf{g}^*$ which we'll return if it generates G .

The first modification we'll try is

if $< \mathbf{g} > = \mathbf{G}$:

return $\mathbf{g}$

If N is abelian, then we try all the $\mathbf{g}^*$ of form $\{g_1, g_2 \dots g_{i-1}, g_i n_j, g_{i+1} \dots g_l\}$ for some i, n_j.

If none of them work, then we are guaranteed that $\mathbf{g} \cup \{n_0\}$ will work, for any $n_0 \in N$.

if $\mathbf{N}$ is an abelian group:

for $1 \leq i \leq l$ and $1 \leq j \leq k$:

$\mathbf{g}^* = \{g_1, g_2 \dots g_{i-1},\ g_i\, n_j\ , g_{i+1} \dots g_l\}$

if $< \mathbf{g}^* > = \mathbf{G}$:

return $\mathbf{g}^*$

return $\mathbf{g} \cup \{n_0\}$

Then, we try out all the generating sets of form $\mathbf{g}^* = \{g_1N_1, g_2N_2 \dots g_tN_t, g_{t+1} \dots g_l\}$ for some $t, N_1, N_2 \dots N_t$.

for $1 \leq t \leq l$:

for any (not necessarily distinct) elements $N_1, N_2 \dots N_t \in \mathbf{N} - \{e\}$:

$\mathbf{g}^* = \{g_1N_1, g_2N_2 \ldots g_tN_t, g_{t+1} \ldots g_l\}$

if $<\mathbf{g}^*> = \mathbf{G}$:

return $\mathbf{g}^*$

Finally, we try out all $\mathbf{g}^* = \{g_1N_1, g_2N_2 \ldots g_tN_t, g_{t+1} \ldots g_l, N_{l+1}\}$ for some $t, N_1, N_2 \ldots N_t, N_{t+1}$

for $1 \leq t \leq l$:

for any elements $N_1, N_2 \ldots N_t, N_{l+1} \in \mathbf{N} - \{e\}$:

$\mathbf{g}^* = \{g_1N_1, g_2N_2 \ldots g_tN_t, g_{t+1} \ldots g_l, N_{l+1}\}$

if $<\mathbf{g}^*> = \mathbf{G}$:

return $\mathbf{g}^*$

At this stage, we are guaranteed that the algorithm must have returned.

Implementation

```
def minimum_generating_set(group) -> list:
    """
    Return a list of the minimum generating set of ``group``.

    INPUT:

    - ``group`` -- a group

    OUTPUT:

    A list of GAP objects that generate the group.

    .. SEEALSO::

        :meth:`sage.categories.groups.Groups.ParentMethods.minimum_generating_set`

    ALGORITHM:

    We follow :doi:`10.1016/j.jalgebra.2023.11.012` (:arxiv:`2306.07633`).

    TESTS:

    Test that the resultant list is able to generate the original group::

        sage: from sage.groups.libgap_mixin import minimum_generating_set
        sage: p = libgap.eval("DirectProduct(AlternatingGroup(5),AlternatingGroup(5))")
        sage: s = minimum_generating_set(p); s
        [(3,4,5)(8,9,10), (1,2,3,4,5)(6,7,8)]
        sage: set(p.AsList()) == set(libgap.GroupByGenerators(s).AsList())
        True

    Test that elements of resultant list are GAP objects::

        sage: from sage.groups.libgap_mixin import minimum_generating_set
        sage: G = PermutationGroup([(1,2,3), (2,3), (4,5)])
        sage: s = minimum_generating_set(G); s
        [(2,3), (1,3,2)(4,5)]
        sage: s[0].parent()
        C library interface to GAP
    """
    if not isinstance(group, GapElement):
        try:
```

```
            group = group.gap()
        except (AttributeError, ValueError, TypeError):
            raise NotImplementedError("only implemented for groups that can construct a gap g

    if not group.IsFinite().sage():
        raise NotImplementedError("only implemented for finite groups")

    # A function to check if the group is generated by the given generators or not.
    def is_group_by_gens(group, gens):
        return set(group.AsList()) == set(libgap.GroupByGenerators(gens).AsList())

    group_elements = group.AsList()

    if group.IsCyclic().sage():
        for ele in group_elements:
            if is_group_by_gens(group, [ele]):
                return [ele]

    if group.IsSimple().sage():
        n = len(group_elements)
        for i in range(n):
            for j in range(i+1, n):
                if is_group_by_gens(group, [group_elements[i], group_elements[j]]):
                    return [group_elements[i], group_elements[j]]

    N = group.MinimalNormalSubgroups()[0]
    n = N.SmallGeneratingSet()

    phi = group.NaturalHomomorphismByNormalSubgroup(N)
    GbyN = phi.ImagesSource()
    GbyN_mingenset = minimum_generating_set(GbyN)

    g = [phi.PreImagesRepresentative(g) for g in GbyN_mingenset]
    l = len(g)

    if N.IsAbelian().sage():
        if is_group_by_gens(group, g):
            return g

        for i in range(l):
            for j in range(len(n)):
                temp = g[i]
                g[i] *= n[j]
                if is_group_by_gens(group, g):
                    return g
                g[i] = temp

        return g + [n[0]]

    # A function to generate some combinations of the generators
    # of the group according to the algorithm. Here it is considered that
    # the first element of the group N is the identity element.
    def gen_combinations(g, N, l):
        iter = product(N, repeat=l)
        for n in iter:
            yield [g[i] * n[l-i-1] for i in range(l)]

    N_list = list(N.AsList())
```

```
    for raw_gens in gen_combinations(g, N_list, l):
        if is_group_by_gens(group, raw_gens):
            return raw_gens

    for raw_gens in gen_combinations(g, N_list, l):
        for nl in N_list[1:]:
            if is_group_by_gens(group, raw_gens+[nl]):
                return raw_gens + [nl]

    assert False
```