Group Algebra Matrix Representations

Zachary Dubinsky

August 13, 2021

Abstract

During my research I implemented algorithms to produce matrix representations of the group algebra $\mathbf{K}[\mathbf{G}]$. The first step was implementing algorithms to produce matrix representations of the group $\mathbf{G}$. To produce the group algebra each matrix representation within the group algebra was scaled by its coefficient and summed to produce a representation for $\mathbf{K}[\mathbf{G}]$. In Section 1, there are two definitions that prove useful. In Section 2 I discuss my first attempt which are tested and proven algorithms that producing matrix representations of $\mathbf{G}$, and then elements of $\mathbf{K}[\mathbf{G}]$. Section 3 goes over algorithms that pursue a more efficient technique using a different data structure. These algorithms were also implemented using the STXXL extended memory library. Finally, Section 4 explains the finer points of how STXXL was and should be used, as well as providing direction for improvements in the future.

1 Definitions

Definition 1. *We define* $n = |\mathbf{G}|$ *where,*

$$|\mathbf{G}| = |\mathbf{H}| \times |\boldsymbol{Sym}(U)| = U! \times m^{U \times k}.$$

This is to clean up messy efficiency notations, though it is important to keep in mind that any efficiency in terms of n is highly inefficient.

Definition 2. *Let* $0 \leq i, j \leq |\mathbf{G}|$ *and* $g_i, g_j \in \mathbf{G}$*. Let (i,j) denote a row-and-column pair where i is the column and j is the row. We define a matrix representation of some* $g \in \mathbf{G}$ *as a* $|\mathbf{G}| \times |\mathbf{G}|$ *matrix with*

$$g_j = g \times g_i \implies (i, j) = 1$$
$$g_j \neq g \times g_i \implies (i, j) = 0.$$

For example, if $g_0 \in \mathbf{G}$ *is the identity then the below matrix illustrates a matrix representation of* $g_2 \in \mathbf{G}$*.*

$$\begin{bmatrix} & 0 & 1 & 2 & ... & n \\ 0 & 0 & 0 & 0 & ... & 0 \\ 1 & 0 & 0 & 0 & ... & 0 \\ 2 & 1 & 0 & 0 & ... & 0 \\ ... & 0 & 0 & 0 & ... & 0 \\ n & 0 & 0 & 0 & ... & 0 \end{bmatrix}$$

2 Original Matrix Representation

The algorithms below are implemented in new_matrix.cpp, and tested in test_new_matrix.cpp.

2.1 G Group Matrix Representation

PRODUCE G REPRESENTATION(g-elem $\in \mathbf{G}$)

```
1  map enum = ⟨g ∈ G, N⟩
2  matrix m = create-zero-matrix(|G|,|G|)
3  row = 0
4  col = 0
5  for g∈ G:
6        row = enum.find(g-elem × g).value()
7        m[row][col] = 1
8        col++
9  return m
```

On line 1 we initialize a mapping that enumerates elements of g. On line 2 we initialize a matrix filled with zeros, with row and column dimensions equal to the $|\mathbf{G}|$. Lines 3 and 4 initialize row and column variables. On line 5, we enter a for loop iterating over all elements of G. On line 6 of the loop, we multiply the provided g-elem with the current g∈G. The map is searched to find the order of the produced g element which is stored in row. We then insert the value 1 into the matrix at the row and column indicated by the variables row and col. Finally, col is incremented by 1 on line 8. The resulting matrix thus aligns with the properties of **Definition 2**, so the algorithm is correct.

2.2 Analysis

Since we are iterating over each g∈ $\mathbf{G}$, the line 5 loop has $\Theta(n)$. Enumerating each element of $\mathbf{G}$ into a map doubles this efficiency, searching the map multiplies the efficiency by a logarithmic factor, we must multiply elements of $\mathbf{G}$, and in the actual implementation the map relies on a method that transforms elements of $\mathbf{G}$ to and from its index representation. These algorithms are $\mathcal{O}(kU^2)$. The total efficiency is roughly

$$\mathcal{O}(2kU^2 \times n)$$

However, the problem becomes insurmountable when the issue of memory is considered. In the actual implementation we require an map that enumerates all elements of $\mathbf{G}$, and a matrix with n^2 entries. In practice, there is not enough memory for most systems to accommodate even the mapping for U,k,m = 5. This led me to investigate a new strategy for producing matrix representations, one we will explore in Section 3. There is an implemented algorithm for producing representations of elements of $\mathbf{K}[\mathbf{G}]$ using the strategy below.

2.3 K[G] Group Matrix Representation

By definition of a group algebra we have that for any k∈ $\mathbf{K}[\mathbf{G}]$,

$$k = c_1 g_1 + c_2 g_2 + ... + c_n g_n$$

for some constants $c_i \in \mathbf{C}$ with $1 \leq i \leq n$, though this representation strategy is only implemented for $c_i \in \mathbf{R}$.

PRODUCE KG REPRESENTATION(k-elem $\in \mathbf{K}[\mathbf{G}]$)

```
1  matrix m = create-zero-matrix(|G|,|G|)
2  for k∈KG:
3        matrix g = PRODUCE G REPRESENTATION(k.g)
4        g ×= k.coef
5        m += g
6  return m
```

Since we may represent any element of G as a matrix using the above algorithm, we may scalar multiply these matrix representations and sum them all to produce a matrix for our given k-elem$\in \mathbf{K}[\mathbf{G}]$.

3 Map Representations

In an effort to reduce the space consumption of the matrix representations I used a map in my final implementation effort. This can be found in exp_extended_mem.cpp. This implementation was also my attempt at an extended memory algorithm. The first algorithm here devises the order of an element of **G**. This was required for the extended memory implementation as data structures in the STXXL library can only store primitive types, and the original enumeration map was not suitable for larger groups.

3.1 G Map Representation

FIND G INDEX(g-elem $\in \mathbf{G}$)

```
1  n = 0
2  for g∈ G:
3        if g == g-elem:
4              return n
5        n++
```

In the actual implementation an iterator is used that generates all elements of a **G** group. The iterator is determined by the U, k, and m data fields which can be found in the **H** group data field of g-elem. The iterator always returns in the same order and contains every element of **G**, so the algorithm will return the correct order of an element in **G**.

PRODUCE G REPRESENTATION(g-elem $\in \mathbf{G}$)

```
1  Let map = ⟨N, N⟩ be an stxxl map.
2  row = 0
3  col = 0
4  for g∈ G:
5        row = FIND G INDEX(g-elem × g)
6        map.insert((row×col)+row,1)
7        col++
8  return map
```

This aligns with **Definition 2** where, for a given g-elem$\in$ **G** and some $g_j \in$ **G**i,

$$g_j = \text{g-elem} \times g_i$$

has that a one is stored in (i,j) of the matrix. This is an improvement as we need not store any zeros. To devise the row and column value from an index, x, let n = |**G**|. Then,

$$\text{column} = x \ \textbf{mod} \ \ n$$

$$\text{row} = \frac{x - \text{column}}{\text{n}}.$$

Graphically the mapping of an index, x, to a row and column is shown below.

$$\begin{bmatrix} & 0 & 1 & 2 & ... & n-1 \\ 0 & 0 & 1 & 2 & ... & n-1 \\ 1 & n & n+1 & n+2 & ... & n+(n-1) \\ 2 & 2n & 2n+1 & 2n+2 & ... & n+(n-1) \\ ... & ... & ... & ... & ... & ... \\ n-1 & (n-1)n & (n-1)n+1 & (n-1)n+2 & ... & (n-1)n+(n-1) \end{bmatrix}.$$

Note that since we are indexing from 0, (n-1) corresponds to n = |**G**| in this context. This design decision was made to allow indexing from 0 as is common in programming.

3.2 Analysis

The first algorithm that finds the order of a **G** element is $\mathcal{O}(n)$ since it iterates over at most every element of **G**. The second algorithm that produces the representation has a $\Theta(n)$ for loop on line 3. The loop multiplies elements of **G**, and utilizes Find G Index to find the order of the product. By definition of group closure each element will be found exactly once. The efficiency of searching for the order of every element in **G** is roughly,

$$\Theta(1 + 2 + ... + n) = \Theta(\frac{n(n-1)}{2})$$

by definition of a geometric series. We must also multiply this efficiency by the time required to multiply elements of **G** and the time required to insert elements into the map. So this algorithm is not nearly as time-efficient as my original implementation, however, there is an improvement on space-efficiency. A matrix representation of an element of **G** contains n^2 entries, but only n of those entries hold the number one, as opposed to zero. The map representation only holds those entries with a one, so the space improvement is significant.

3.3 K[G] Map Representation

Using the **G** group map represented above, the following algorithm produces matrix representations of the group algebra, held in map structures.

```
PRODUCE KG REPRESENTATION(kg ∈ K[G])
1  map = ⟨N, N⟩
2  g-map = ⟨N, N⟩
3  for k∈ K[G]:
4       g-map = PRODUCE G REPRESENTATION(k.g)
5       for K∈ g-map = ⟨K,V⟩:
6            map.insert(K,k.coef)
7  return map
```

Let K be the key corresponding to row = r and column = c. Then for some $g \in \mathbf{G}$,

$$g_r = g \times g_c.$$

If there exists some $g_0 \in \mathbf{G}$ where $g_r = g_0 \times g_c$ then $g_0 = g$ by the definition of a group. So two distinct matrix representations will not contain an entry in the same row and column, or an entry other than one. Thus, we may explicitly insert the coefficient of the $\mathbf{G}$ element into the $\mathbf{K}[\mathbf{G}]$ map at that matrix index.

3.4 Analysis

Let $k = c_0g_0 + c_1g_1 + ... + c_ng_n$ be a group algebra over $\mathbf{G}$. Let m be the number of $\mathbf{G}$ elements with non-zero coefficients c_i. Then the loop on line 3 is $\Theta(w)$. Line 4 is $\Theta(\frac{n(n-1)}{2})$ as shown above, and the line 5 loop is $\Theta(n)$. Within the loop on line 5, the insertion and retrieval operations are logarithmic, and the multiplication and addition operations are constants. As such, the algorithm itself is roughly

$$\Theta(w \times \frac{n(n-1)}{2} + n)$$

$$\vdots$$

$$= \Theta(w \times \frac{n^2+n}{2})$$

and in the worst case, representing $c_1g_1 + c_2g_2 + ... + c_ng_n = k \in \mathbf{K}[\mathbf{G}]$ where $0 < c_1...c_n$, so $w = n = |\mathbf{G}|$, the algorithm has

$$\mathcal{O}(\frac{n^3+n^2}{2}).$$

4 Extended Memory Matrix Representation

Storing even the enumerative mapping for $5 \leq$ U,k,m in memory was not feasible on my system. It required storing $U! \times m^{U \times k}$ elements, each of which were a Standard Template Library pair. This pair contained an 8 byte integer and a vector of 8×U byte integers. The matrix was even bigger with the original strategy. This lead me to investigate libraries where data structures could be held in storage, rather than memory. The STXXL library implements data structures and algorithms that perform computations in storage, and mirrors the Standard Template Library in its syntax. The map algorithms in Section 3 have all been implemeneted in exp_extended_mem.cpp using the STXXL library.

4.1 Hurdles

The STXXL library comes with a few implementation hurdles. First and foremost, its data structures can only store primitive types like int, char, float, double, and boolean. Types like vectors or strings will not work as they contain pointers to memory addresses, thus undermining the intention to maintain data in storage. Storing STXXL structures in other STXXL structures will not work either. The other significant issue is that the STXXL map, and many other structures in the library, inherit from noncopyable. I overcame this issue by using pointers to the map which can be passed between functions.

4.2 Improvements and Next Steps

The algorithm to find the index of a **G** element creates significant inefficiencies. In my original implementation I mapped elements of the group to their enumeration, however the map could not accommodate groups of a sufficiently large size, and this strategy cannot be explicitly transferred to the STXXL map as it only stores primitive types. Work is needed to efficiently determine the order of an element of **G**.

In addition, the algorithm uses the long double type to enumerate the **G** group, and to represent row and column entries in the map version of the matrix. There is a maximum length to this type, and the size of groups grow at a factorial rate. As such, finding the order of a group will not be effective for sufficiently large **G**.

The Section 2 matrix representation algorithms are implemented using the new_matrix class, so all matrix operations are implemented. The Section 3 map representation algorithm have matrix multiplication and addition implemented, though they are not efficient. Moreover, the Section 2 algorithm to find the order of an element of **G** is highly inefficient. This will need improvement to make the algorithms usable.

Finally, I encountered the TPIE library which implements extended memory data structures and algorithms.This library still receives attention from developers, whereas the STXXL library has not been updated in years, so this could be worth investigation.