Folding Algorithm

Param Gujral

December 28, 2020

1 Introduction

In this paper we describe an algorithm that takes a subgroup $H = < g_1, g_2, ... >$ where g_i is an element of a free group and calculates the basis of H. Our algorithm uses the concept of folding in graphs that is explained in the book Office Hours with a Geometric Group Theorist.

2 Folding Algorithm

The algorithm that was implemented on BlueJ, a Java IDE, had essentially two parts:

Program 1: The first program took the generators g_i as input and created a graph G with petals: one petal for each generator g_i centred at the base vertex. This construction ensures that every loop at the base vertex can be written in terms of the generators g_i. The program takes this graph G and keeps folding edges successively till it reaches an immersion Δ_H. The fundamental group of this immersion is isomorphic to the subgroup H.

The implementation of the first program:

Step 1 (Lines 13 to 27): Accepts the generators of H as input and stores each generator as a separate element in an array of Strings.

Step 2 (Lines 29 to 41):

Calculates the number of vertices in the original graph from the generators entered by essentially summing the lengths of the strings entered and subtracting from it the number of generators and adding 1.

Step 3 (Lines 49 to 116):
Fills the two main arrays $vert1[][]$ and $vert2[][]$ which store the a and b edges of the graph G in an adjacency matrix format. "0" indicates there is an edge between the two vertices and "-1" indicates there isn't. We use the variables

$tback$ and $tforw$ to keep track of the current and the next vertex while creating petals. (Using arrays to store edges implicitly performs $type2$ folds across vertices.)

Step 4 (Lines 150 - 435):
Checks for the existence of a fold. This step in its entirety is contained in an infinite while loop. There are 4 conditions to check for $Type1$ folds:

- $vert1[i][j_1] = 0$ and $vert1[i][j_2] = 0$: two a edges beginning at i
- $vert2[i][j_1] = 0$ and $vert2[i][j_2] = 0$: two b edges beginning at i
- $vert1[j_1][i] = 0$ and $vert1[j_2][i] = 0$: two a edges entering i
- $vert2[j_1][i] = 0$ and $vert2[j_2][i] = 0$: two b edges entering i

For the first condition, we use the variables $flag1$, its boolean value indicating whether such a fold is possible, and $c1$ and $c2$, standing in for j_1 and j_2 respectively. Similar variables are used for the other conditions.

The breaking condition (lines 230-236): if all the $flag$ variables are false then we know that no folds are possible and that we have reached an immersion.

Switching the edges (Lines 245-285, 290-324, 327-363, 367-401): If a fold is indeed possible then by convention we collapse the $c1$ vertex. All the edges entering or leaving $c1$ are reoriented to $c2$ for the first condition. The switching process is essentially the same for the four conditions.

Step 4 keeps running till the breaking condition is satisfied.

Step 5 (Lines 450-465):
After the while loop breaks, the immersion is printed.

Program 2. The second program takes the immersion that was calculated by the first program and outputs the basis of the fundamental group of the immersion. This is also the basis of the subgroup H.

The basis is computed by identifying the maximal tree in the graph. For each edge not in the maximal tree it draws a loop in the maximal tree that passes the edge. Each loop becomes a basis element for the fundamental group of that graph. Since the fundamental group of the immersion is isomorphic to H, we get a basis for H. (The maximal tree is identified by adapting a standard Breadth First Search Java implementation taken from www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph.)

The implementation of the second program:

Step 1 (Lines 95-155):
Representing the *aedges* of the immersion in $vert1$ and *bedges* in $vert2$ like before.

Step 2 (Carried out during Step 1):
As the edges are getting stored, the edges are simultaneously being added to a linked list for the Breadth First Search. For instance, if there is an *aedge* from $aedge1$ to $aedge2$ then we must add $aedge1$ to the linked list associated to $aedge2$ and vice versa.

We must add each edge twice because in a traditional Breadth First Search the convention is to restrict movement along a directed edge.

Step 3 (line 159, then 56 to 81):
When g.BFS(0), is reached at line 159 The Breadth First Search is carried out. To suit our purposes, we added the array $EdgesBFS$. During the traversal, if we are currently at the vertex s and visit the vertex n, then we consider the edge (s, n) to be in the maximal tree.

Step 4 (lines 163 to 186):
In Step 2 we had mentioned that if there is say an *aedge* from $aedge1 = 5$ to $aedge2 = 6$ we must add each one to the other's linked lists even if there isn't an edge from vertex 6 to vertex 5. As a result, it is possible that during the Breadth First traversal that vertex 5 is visited by vertex 6 so the edge $(6, 5)$ is identified in $EdgesBFS$ to be part of the maximal tree even though there is neither an *a edge* nor a *b edge* from vertex 6 to 5.

To remedy this, we created the arrays $vert3$[][] and $vert4$[][]. (These arrays were initialised to "-1.") In this step, we sift through $EdgesBFS$. If say $EdgesBFS$[6][5] = 0, indicating that the edge (6,5) is in the maximal tree we check if $vert1$[6][5] = 0 or $vert2$[6][5] = 0. If this condition is true, there is nothing to be done.

If however (6,5) has neither an *a edge* nor a *b edge* then check if $vert1$[5][6] = 0 or $vert2$[5][6] = 0. This second condition must be true. Then if $vert1$[5][6] = 0 we change $vert1$[5][6] to "-1" and $vert3$[6][5] to "0". Instead, if $vert2$[5][6] = 0 we change $vert2$[5][6] to "-1" and $vert4$[6][5] to "0." We perform either one of the changes but not both.

$vert3$ stores the induced $a^{-}1$ *edge* and $vert4$ stores the induced $b^{-}1$ *edge* only in this special case of a maximal edge being neither an *a edge* nor a *b edge*.

Step 5 (lines 190-280):
This step calculates the generator corresponding to each edge that is not in the maximal tree. This generator is a loop that travels through the maximal tree and then moves through the edge and back through the tree to the base.

(Lines 196-220): The program checks if there are any “extra“ edges that are not in the tree.

(Lines 230-280): If there are any “extra“ edges, then we go trace the path backwards from both endpoints to the base vertex. For tracing this path backwards we use $EdgesBFS$. For each edge that we trace backwards, we add the corresponding element to our loop.

(Lines 285-340): For each “extra“ edge we create a generator using the path we traced out.

Step 6 (Lines 346-350):
Finally, the string containing the generators is printed. This is the basis of the subgroup H.

3 Conclusion

Both the programs together compute the basis of the subgroup H of a free group presented with its generators.