Gordian: User Manual

Loren Abrams Blake Mellor Lowell Trott

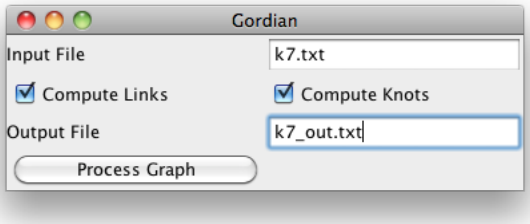
June 4, 2013

Abstract

Gordian is a program to count the links and knots in a spatial graph. Links are detected using the linking number; knots are detected using the second coefficient of the Conway polynomial. The program is written in Java. This manual describes how to use the program – most importantly, it describes the required format for the input.

Running the program

When *Gordian* is run, a window will appear asking for an input file and an output file, as well as whether the user wants to count knots, links or both. The window (on a Macintosh) appears as shown:



The user will write in an input file and output file (in this case, the files were `k7.txt` and `k7_out.txt`). These should be plain text files; the program will overwrite the contents of the output file, if it already exists. The files should be in the same folder as the *Gordian* program. The checkboxes for counting links and knots are both checked by default; the user may uncheck them if desired. Finally, click on the "Process Graph" button to perform the calculation.

Input and Output

We begin with an example. Consider the graph diagram in Figure 1. The input file for this diagram can be written in two ways – both are shown in Figure 2. The output from the program is shown in Figure 3.

The first line indicates the number of vertices in the graph. It must be in the form `size = n`, where `n` is the number of vertices. The second line indicates whether the file will specify edges that should be *added* to the empty graph (with no edges), or edges that should be *removed* from the complete graph on these vertices. The second option is convenient when, as in this example, the graph is almost complete.

The subsequent lines contain the edges to be added (or removed) and the crossing information. Each line should contain one edge or one crossing; the edges and crossings may be in any order (i.e. the edges may come first, or the crossings may come first, or they can be intermixed). To determine the edges and crossing information, you must first choose a labeling of the vertices of your graph. If the graph has n vertices, the labels should range from 0 to $n - 1$.

Edges

To enter an edge, enter the pair of labels for the endpoints of the edge. The pair may be entered in either order.

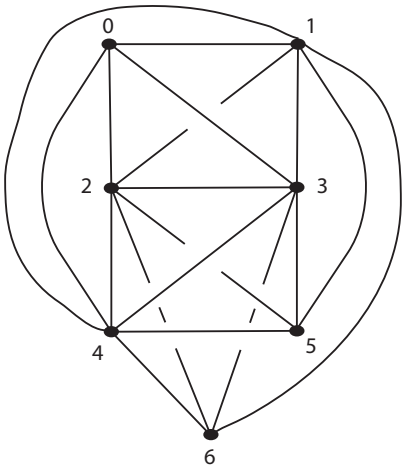


Figure 1: A diagram of $K_{3,1,1,1,1}$

First input file:

```
size = 7
add edges
0,1
0,2
0,3
0,4
1,2
1,3
1,4
1,5
1,6
2,3
2,4
2,5
2,6
3,4
3,5
3,6
4,5
4,6
0,3,1,2,1,1,-1
3,4,2,5,1,1,1
3,4,2,6,2,1,1
4,5,2,6,1,2,-1
2,5,3,6,2,1,-1
4,5,3,6,2,2,-1
1,4,0,6,2,1,-1
0,5,1,4,1,1,1
0,5,1,6,2,1,1
```

Second input file:

```
size = 7
remove edges
0,5
0,6
5,6
0,3,1,2,1,1,-1
3,4,2,5,1,1,1
3,4,2,6,2,1,1
4,5,2,6,1,2,-1
2,5,3,6,2,1,-1
4,5,3,6,2,2,-1
1,4,0,6,2,1,-1
0,5,1,4,1,1,1
0,5,1,6,2,1,1
```

Figure 2: Input files for $K_{3,1,1,1,1}$

Output file:

```
Gordian's Input:
size = 7;

edges:
0,1
0,2
0,3
0,4
1,2
1,3
1,4
1,5
1,6
2,3
2,4
2,5
2,6
3,4
3,5
3,6
4,5
4,6

crossings:
0,3,1,2,1,1,-1
3,4,2,5,1,1,1
3,4,2,6,2,1,1
4,5,2,6,1,2,-1
2,5,3,6,2,1,-1
4,5,3,6,2,2,-1
1,4,0,6,2,1,-1
0,5,1,4,1,1,1
0,5,1,6,2,1,1

Your graph has the following 0 knots:

Your graph has the following 3 links:
{ 1, 2, 5 } { 3, 4, 6 }  lk = 1
{ 1, 2, 5 } { 0, 3, 6, 4 }  lk = -1
{ 3, 4, 6 } { 0, 1, 5, 2 }  lk = -1
```

Figure 3: Output file for $K_{3,1,1,1,1}$

Crossings

A crossing is a 7-tuple `a, b, c, d, m, n, k`, where:

`a` and `b` are the endpoints of the *overcrossing* edge.

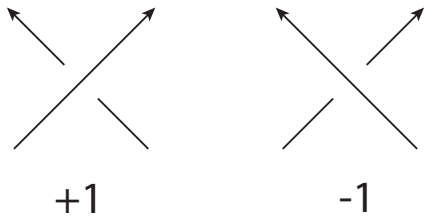
`c` and `d` are the endpoints of the *undercrossing* edge.

`m` is the order of this crossing along edge `(a,b)`, counting from `a` to `b`.

`n` is the order of this crossing along edge `(c,d)`, counting from `c` to `d`.

`k` is the sign of the crossing, assuming the edges are oriented from `a` to `b` and from `c` to `d`.

The sign of the crossing is determined by the standard right-hand rule, as shown below:



Output

The output file first recapitulates the input. Then it provides the number of knots with a list of each knotted cycle, together with the value of $a_2(K)$ for each knotted cycle K. Finally it provides the number of links, with a list of each pair of linked cycles, together with their linking number. The signs of the linking numbers and a_2's will depend on the labeling of the original graph.

Procedures for computing the number of links and knots in a spatial graph

In this section we will describe the algorithms used to count the number of links and knots in a spatial graph.

Representing the graph and finding the cycles

The first task is to represent a particular graph embedding in a form usable by the program. The user supplies the number of vertices in the graph, the edges of the graph, and information about each crossing in the graph (the details of the input format are described in the user's manual at *http://myweb.lmu.edu/bmellor/Gordian*. The vertices of the graph are numbered from 0 to $n-1$ (for a graph with n vertices), and the abstract graph is stored as an adjacency matrix (i.e. an array) `int[n][n] adjacent`, where `adjacent[i][j]` contains the value 1 if vertices `i` and `j` are connected by an edge, and the value 0 otherwise.

We also need to record the details of the particular spatial embedding of the graph; specifically, how the edges are crossing each other. We assume each edge is oriented from its smaller endpoint to its larger endpoint (i.e. if `i` and `j` are adjacent vertices with $i < j$, then the edge is oriented from `i` to `j`). If there are k crossings, then the crossing information is contained in another array `int[k][7] crossings`. For crossing `i` we have:

- `crossings[i][0]` and `crossings[i][1]` are the endpoints of the edge that crosses *over* the other edge.
- `crossings[i][2]` and `crossings[i][3]` are the endpoints of the edge that crosses *under* the other edge.
- `crossings[i][4]` gives the order of the crossing among all the crossings along the over-crossing edge (following the orientation of the edge).
- `crossings[i][5]` gives the order of the crossing among all the crossings along the under-crossing edge.
- `crossings[i][6]` gives the sign of the crossing ($+1$ or -1).

We are interested in counting the knotted and linked cycles in the graph, so the next step is to generate all the cycles in the graph. This is done by going through all possible cycles (i.e. all cycles in the complete graph on the given vertices), and then removing any cycles which contain an edge not contained in the adjacency matrix for the graph. Each cycle is given an (arbitrary) orientation determined by the order of the vertices; this may not agree with the default orientation on each edge.

Counting linked cycles

To count the linked cycles, the program computes the linking number for each pair of disjoint cycles. This requires counting the crossings between each pair of edges (with sign). For efficiency, the program first extracts this data from the `crossings` array, constructing an array `int[n][n][n][n] crossingMatrix`, where `crossingMatrix[a][b][c][d]` is the sign of the crossing between edges (a, b) and (c, d), oriented from a to b and from c to d. So `crossingMatrix[b][a][c][d]` is the negative of `crossingMatrix[a][b][c][d]`, and so forth.

The program then compares each cycle to every other cycle, and follows the following procedure for each pair of cycles:

1. If the cycles share any vertices, then the pair is not disjoint; move on to the next pair.
2. Compare each edge of the first cycle to each edge of the second cycle, and add up their crossing numbers (from `crossingMatrix`); use the orientation for each edge induced by the orientation of the cycle.
3. The sum is the linking number. If the linking number is nonzero, add this pair to the list of links, and increase the number of links by 1.
4. Move on to the next pair of cycles.

The result is a list of all pairs of cycles with non-trivial linking number, and the number of such pairs.

Counting knotted cycles

The program identifies knotted cycles using the second coefficient of the Conway polynomial, a_2. To compute a_2, we apply the skein relation:

$$a_2(K_+) - a_2(K_-) = \mathrm{lk}(L_0)$$

where K_+, K_- and L_0 are identical except in a neighbourhood of a single crossing, where they differ as shown in Figure 4. Observe that if K_+ and K_- are knots (differing by a single crossing change), then L_0 is a link of two components.

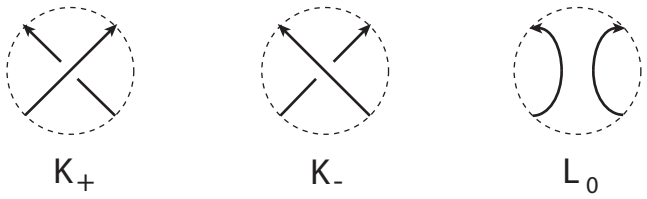


Figure 4: The diagrams K_+, K_- and L_0.

It is well known that given a diagram of an oriented knot K and a starting point p on the knot, we can change K to a diagram of the unknot by changing crossings so that as we traverse K, starting at p, the first time we encounter each crossing we cross *over* the other strand. Since $a_2(\text{unknot}) = 0$, this means we can find a sequence of knots $K = K_0, K_1, K_2, \ldots, K_m =$ unknot such that K_i and K_{i+1} differ by a single crossing change. So:

$$a_2(K_i) = a_2(K_{i+1}) \pm \mathrm{lk}(L_i) \implies a_2(K) = \sum_{i=0}^{m-1} \epsilon_i \mathrm{lk}(L_i)$$

where $\epsilon_i = \pm 1$.

This provides a procedure for computing a_2 for each cycle in the spatial graph. For efficiency, we begin by constructing `boolean[n][n][n][n] overMatrix`, where `overMatrix[a][b][c][d]` is `true` if edge ab crosses over edge cd, and `false` otherwise. Similarly, we construct a `crossingOrderMatrix` which records for each edge the other edges that cross it, and their order. Now, for each cycle, we proceed as follows:

1. Begin traversing the cycle. At each edge, go through the crossing for that edge (from the `crossingOrderMatrix`), and determine if any are places where the cycle crosses itself. If there is a self-crossing, check whether it is an over- or under-crossing, and whether it has been encountered before. If it is an over-crossing, or has been encountered, move on to the next crossing.
2. If the crossing is an under-crossing that has not been encountered, change it (temporarily) to an over-crossing. Then use the order of the crossings along the edges to construct the two links in L_i, and compute their linking number as described in Section . Add the result (with appropriate sign) to the running total for a_2.

3. When all the crossings have been considered, check whether $a_2 = 0$. If it is non-zero, then the cycle is a knot, and we add it to our list of knotted cycles, and increase the total number of knots by 1.

4. Undo any changes made to the `crossingMatrix`, `overMatrix`, `crossingOrderMatrix`, etc. Then move on to the next cycle.

The result is a list of cycles that have non-zero a_2, and the number of such cycles.