C Computation of Tutte Polynomials

Michael Barany
advised by Professor Victor Reiner

University of Minnesota Undergraduate Research Opportunities Program

February, 2005

Contents

Abstract

This documentation accompanies the programs `tutte.c`, `tutte_modp.c`, and `tutte_th.c`, accessible via Professor Victor Reiner's homepage. The program computes the Tutte polynomial for a matroid. The code is in standard `C`, so in theory it can be used on platforms other than Linux, but that is the only one for which it has been tested extensively. The manual is intended primarily for Linux users, but I'm sure if you're really really clever you can make the program work on another machine. This project was supported by the University of Minnesota Undergraduate Research Opportunities Program.

Chapter 1

Summary

1.1 Overview

All three programs compute the Tutte polynomial for a matroid from file through a direct deletion-contraction algorithm. `tutte.c` does this for an integer matroid. `tutte_modp.c` completes this computation for a matroid in $\mathbb{Z}/p\mathbb{Z}$. `tutte_th.c` is included as an alternative if there are overflow problems in `tutte.c`, and will be explained in the "Algorithm"section.

1.2 Compiling

In a terminal window, type:

```
[prompt]$ gcc tutte.c -o tutte.out   #replace tutte.out with any output name
```

All three programs should compile in a second or two and will be deposited in the current directory.

1.3 Syntax

The syntax to run the program is as follows:

```
[prompt]$ ./tutte.out d n matrix.dat
```

The first term, `tutte.out`, is the name of the output file to which you compiled the source code. `d` and `n` are, respectively, the dimension of the matroid and the number of hyperplanes. Some useful synonyms:

`d` dimension, number of rows, $|V(G)|$ for a graph G
`n` number of hyperplanes, number of columns, $|E(G)|$ for a graph G

`matrix.dat` is an input file containing the matrix to be computed. It could have whatever name you please. It might even be in a subdirectory, as in:

```
[prompt]$ ./tutte.out 7 21 matrices/k7.dat
```

You can also redirect the output by typing something like the following:

```
[prompt]$ ./tutte.out 7 21 matrices/k7.dat >TutteofK7
```

This will store the output in a file named `TutteofK7`.

For `tutte_modp.c`, run by typing:

```
[prompt]$ ./tutte_modp.out d n p matrix.dat
```

Where `p` is the prime divisor in $\mathbb{Z}/p\mathbb{Z}$.

`tutte_th.c` (I won't go into more detail here) is run by typing:

```
[prompt]$ ./tutte_th.out d n th matrix.dat
```

1.4 Input

The input should be in a plain text document entered column by column. The program doesn't look for any particular extension, so any name will do. There must be blank space between each number (enter only integers!). Other than that, just about anything works. For instance, to enter the matrix

$$\begin{bmatrix} 2 & 4 & 1 & 1 & 0 & 2 \\ 0 & 3 & 5 & 7 & 1 & 0 \\ 1 & 0 & 1 & 2 & 0 & 0 \\ 1 & 1 & 0 & 3 & 1 & 0 \end{bmatrix}$$

One could open a file, for instance, by typing:

```
[prompt]$ pico mymatrix.dat
```

and, once in the editor, type:

```
2    0    1    1
4    3    0    1
1    5    1    0
1    7    2    3
0    1    0    1
2    0    0    0
```

or, if you like,

```
2 0 1 1 4 3 0 1 1 5 1 0 1 7 2 3 0 1 0 0 2 0 0 0
```

or, if it strikes your fancy,

```
2
0
1
1
```

```
4
3
0
...
```

Another thing to keep in mind: if you have a graph, you will need to give the edges an orientation. Loops are columns of zeros. The matrix to enter is the incidence matrix, with vertices indexing rows and edges indexing columns.

1.5 Output

All three programs will print the input matrix (so you can check) and then, shortly (or not so shortly) thereafter, the Tutte polynomial. The polynomial is printed in a single line, without any carriage returns. This is great if the output comes out in a display that automatically wraps text (such as Terminal, or an office program), because it's much easier to copy, paste, and manipulate data. If the display truncates the text with an annoying "`$`"character, you'll only see the first few terms of the polynomial. If you're not sure what that all meant, go ahead and read the following two examples.

Chapter 2

Examples

2.1 K_7, an Example

Suppose you want to compute $T_{K_7}(x, y)$, the Tutte polynomial for the complete graph on 7 vertices. First, compile the program (if you haven't already done so):

```
[prompt]$ gcc tutte.c -o tutte.out
```

Second, you'll need to create a file with the matrix information. Open a new data or text file

```
[prompt]$ pico k7.dat
```

type the incidence matrix for K_7 into the file

```
1   -1   0   0   0   0   0
1   0   -1   0   0   0   0
1   0   0   -1   0   0   0
1   0   0   0   -1   0   0
1   0   0   0   0   -1   0
1   0   0   0   0   0   -1
0   1   -1   0   0   0   0
0   1   0   -1   0   0   0
0   1   0   0   -1   0   0
0   1   0   0   0   -1   0
0   1   0   0   0   0   -1
0   0   1   -1   0   0   0
0   0   1   0   -1   0   0
0   0   1   0   0   -1   0
0   0   1   0   0   0   -1
0   0   0   1   -1   0   0
0   0   0   1   0   -1   0
0   0   0   1   0   0   -1
0   0   0   0   1   -1   0
0   0   0   0   1   0   -1
```

```
0   0   0   0   0   1   -1
```

and output it by typing `[ctrl] o` and then `[enter]`. Exit `pico` by typing `[ctrl] x`.

Third, run `tutte.out`

```
[prompt]$ ./tutte.out 7 21 k7.dat
```

(Press `[enter]`.) Now, the screen should look like:

```
[prompt]$ ./tutte.out 7 21 k7.dat

[ 1  1  1  1  1  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  ]
[ -1  0  0  0  0  0  1  1  1  1  1  0  0  0  0  0  0  0  0  0  0  ]
[ 0  -1  0  0  0  0  -1  0  0  0  0  1  1  1  1  0  0  0  0  0  0  ]
[ 0  0  -1  0  0  0  0  -1  0  0  0  -1  0  0  0  1  1  1  0  0  0  ]
[ 0  0  0  -1  0  0  0  0  -1  0  0  0  -1  0  0  -1  0  0  1  1  0  ]
[ 0  0  0  0  -1  0  0  0  0  -1  0  0  0  -1  0  0  -1  0  -1  0  1  ]
[ 0  0  0  0  0  -1  0  0  0  0  -1  0  0  0  -1  0  0  -1  0  -1  -1  ]
```

Soon, to your great delight, it will look like:

```
[prompt]$ ./tutte.out 7 21 k7.dat

[ 1  1  1  1  1  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  ]
[ -1  0  0  0  0  0  1  1  1  1  1  0  0  0  0  0  0  0  0  0  0  ]
[ 0  -1  0  0  0  0  -1  0  0  0  0  1  1  1  1  0  0  0  0  0  0  ]
[ 0  0  -1  0  0  0  0  -1  0  0  0  -1  0  0  0  1  1  1  0  0  0  ]
[ 0  0  0  -1  0  0  0  0  -1  0  0  0  -1  0  0  -1  0  0  1  1  0  ]
[ 0  0  0  0  -1  0  0  0  0  -1  0  0  0  -1  0  0  -1  0  -1  0  1  ]
[ 0  0  0  0  0  -1  0  0  0  0  -1  0  0  0  -1  0  0  -1  0  -1  -1  ]
120 y^1 + 490 y^2 + 945 y^3 + 1225 y^4 + 1260 y^5 + 1120 y^6 + 895 y^7
+ 645 y^8 + 420 y^9 + 245 y^10 + 126 y^11 + 56 y^12 + 21 y^13 + 6 y^14
+ 1 y^15 + 120 x^1 + 644 x^1 y^1 + 1225 x^1 y^2 + 1330 x^1 y^3 + 1085
x^1 y^4 + 756 x^1 y^5 + 469 x^1 y^6 + 245 x^1 y^7 + 105 x^1 y^8 + 35
x^1 y^9 + 7 x^1 y^10 + 274 x^2 + 721 x^2 y^1 + 700 x^2 y^2 + 420 x^2
y^3 + 210 x^2 y^4 + 84 x^2 y^5 + 21 x^2 y^6 + 225 x^3 + 280 x^3 y^1 +
105 x^3 y^2 + 35 x^3 y^3 + 85 x^4 + 35 x^4 y^1 + 15 x^5 + 1 x^6
```

2.2 Another Example in $\mathbb{Z}/3\mathbb{Z}$

Suppose, for whatever reason, you wanted to conpute $T_M(x, y)$ for the matroid of all non-zero three dimensional vectors, up to scaling, in $\mathbb{Z}/3\mathbb{Z}$. As in the previous example, you would enter the following matrix into a data file (such as `Zmod3Z.dat`):

```
[ 1  0  0  1  1  0  2  0  2  1  2  1  1  ]
```

```
[ 0  1  0  1  0  1  1  2  0  1  1  2  1  ]
[ 0  0  1  0  1  1  0  1  1  1  1  1  2  ]
```

Then enter the command-line call:

```
[prompt]$ ./tutte_modp.out 3 13 3 Zmod3Z.dat
```

and to your sudden and intense amusement the following will appear on your screen:

```
[ 1  0  0  1  1  0  2  0  2  1  2  1  1  ]
[ 0  1  0  1  0  1  1  2  0  1  1  2  1  ]
[ 0  0  1  0  1  1  0  1  1  1  1  1  2  ]
16 y^1 + 32 y^2 + 36 y^3 + 28 y^4 + 21 y^5 + 15 y^6 + 10 y^7 + 6 y^8 +
3 y^9 + 1 y^10 + 16 x^1 + 26 x^1 y^1 + 13 x^1 y^2 + 10 x^2 + 1 x^3
```

2.3 Playing with Mathematica

Personally, I enjoy just looking at the output polynomial and contemplating the meaning of deletion-contraction isomorphism invariance. Most users, however, will want to somehow manipulate the output. For this reason, I include here a sample use of Mathematica to process a generated polynomial.

Suppose you want the characteristic polynomial for the matroid in $\mathbb{Z}/3\mathbb{Z}$ discussed above. Copy the polynomial from the Terminal (or other output viewing application) and paste it into Mathematica:

```
In[1]:= 16 y^1 + 32 y^2 + 36 y^3 + 28 y^4 + 21 y^5 + 15 y^6 + 10 y^7 +
6 y^8 + 3 y^9 + 1 y^10 + 16 x^1 + 26 x^1 y^1 + 13 x^1 y^2 + 10 x^2 + 1
x^3
```

Out[1]= $16x+10x^2+x^3+16y+26xy+32y^2+13xy^2+36y^3+28y^4+21y^5+15y^6+10y^7+6y^8+3y^9+y^{10}$

Then, tell it to factor the expression obtained by substituting $1-t$ for x and 0 for y:

```
In[2]:= Factor[%/.{x->1-t,y->0}]
```

Out[2]= $-((-9+t)(-3+t)(-1+t))$

This result is good, because it corresponds with the known characteristic polynomial for such matroids:

$$\prod_{i=0}^{d-1}(p^i-t)$$

2.4 For Mac OS X

To use this program on a computer running Mac OS X, you'll need the developer tools installed. Consult www.mac.com before doing anything, but most of the installation involves double-clicking

Developer.mpkg in the Applications/Installers/Xcode Tools folder. Then all of the above can be done from the command line in the Terminal (Applications/Utilities). The only change that needs to be made is in the source code. Change `<malloc.h>` to `<unistd.h>` near the top of the file. Run-times and performance may differ.

Chapter 3

Algorithm

The goals of this section are:

1. Allow general users to understand the process by which the program does its computations.
2. Allow users familiar with `C` to make modifications without crashing the program.
3. Explain some tools that can be used if you suspect that the output is flawed.
4. Give anecdotal evidence about the speed and efficiency of the algorithm.

3.1 Overview

`tutte.c` uses a simple deletion-contraction depth-first algorithm to compute the Tutte polynomial for a matroid inputted from file. The matrix is put in row-reduced echelon form, making it is easy to detect loops (columns of zeros) and isthmuses (pivot columns in rows with only one non-zero entry). The program progressively contracts the first non-loop, non-isthmus edge it encounters until it reaches a base case of all loops and isthmuses (i.e. a spanning forest with loops allowed). Once it reaches this base case, the program increments the appropriate term of the Tutte polynomial, reconstructs the matrix up to the last contraction, turns the last contraction into a deletion, and continues as above. In this way, the program searches all possible deletion-contraction paths to the set base case. To save computation time, an additional base case was added for single dimension/vertex matroids, whose Tutte polynomials can be inductively determined with ease.

3.2 Run-time

As described above, `tutte.c` tallies entries in the Tutte polynomial one term at a time. This makes the run-time roughly proportional to $T_M(1,1)$, or the complexity of the matroid. Of course, the time to compute each term increases with larger matroids, but this computation time also depends on the complexity of matroid (sparser large matroids will be computed much more quickly than

dense small matroids). For simple graphs of n vertices, $T_M(1,1)$ is bounded by $n^{n-2} = T_{K_n}(1,1)$. K_n runs very quickly for $n \leq 7$, takes under two minutes for $n = 8$, and less than an hour for $n = 9$.

3.3 The Lowest Terms Threshold

In order to keep numbers in integer form, the algorithm for row-reduction involves a great deal of scaling. As a result, numbers can become quite large very quickly. For this reason, the program frequently checks that the largest number in the matrix isn't above a threshold value set at the top of the code to the integer TH. If it exceeds this value, the program then tests each row of the matrix for divisibility by a preset list of primes (see below) and scales them appropriately.

If you suspect that your matroid is not computing correctly due to overflow errors (the most likely source of error, marked by under-counting in the polynomial), or you want to see if you can make it run faster, the program `tutte_th.c` includes the TH value as a command-line argument (see syntax section).

3.4 The Prime List

A similarly easy to modify parameter that can affect results and run-time is the Prime List. Defined at the top of the program, this is the list of factors for which the program tests when it is rescaling rows of the matrix. The factory default includes several primes that the program rarely encounters. In fact, when computing Tutte for K_8 the program never consults the list. Other large matroids divide by two an obnoxious number of times but never seem to agree on an odd number. We tentatively conjecture that the only common factors that appear across rows of the matrix as it is being row-reduced and linearly recombined also appear as determinants of the matrix's principal minors. If you are interested in this phenomenon, you may solicit a variant of the code I wrote to count factorings. If you're worried about run-time or accuracy you can truncate or add to the list appropriately before compiling the code. The program still functions properly if the Prime List is empty, provided that `NUMPRIMES` is set to zero. Otherwise, `NUMPRIMES` must be the number of primes in the Prime List.

3.5 Program Functions, Deconstructed

The goal of this section is to allow an advanced user who is competent in `C` to modify the program to meet their needs and give all users an idea of how the program operates. All functions begin with variable declarations which I will explain on a need-to-know basis. Most non-obvious ones are just tallying variables. Functions are listed in some approximation of a logical order. They appear in the program more or less in the order they were written, but shouldn't be too hard to cross-reference.

3.5.1 `main`

All `C` programs have a main function which is called by the computer and is responsible for sending something back. (In this case, as in most, it returns the number 0.) The function first checks that the command line arguments are in good shape and that it can open the user's matrix. It then creates two copies of that matrix: one as a reference copy and one for actually doing operations. It also allocates a matrix to store the entries of the Tutte polynomial, with the coefficient of $x^i y^j$ stored in the $a_{i,j}$ entry of the matrix. The function also creates a vector that records the sequence of deletions and contractions. If you think of the Tutte computation as traversing a binary tree of deletions and contractions of appropriate edges, this vector records steps as you walk down that tree.

Using the indicator variable `notfinished`, the program runs until the value of `notfinished` is set to 0. It starts by row-reducing the matrix and finding and contracting the first edge that is neither a loop nor an isthmus. The `nextedge` variable serves two functions. When it is greater than 0, it stores the position of the desired edge. When it is 0 or -1, it indicates the two types of base cases that occur when the `justloopsnbridges` function cannot find a valid non-loop, non-isthmus edge or when the dimension of the current matrix is 1.

`main` continues contracting and row-reducing until it hits a base case. Depending on which case it encounters, `main` sends the base case matrix to another function which increments the appropriate value(s) in the Tutte polynomial matrix. `main` then calls another function to rebuild the current matrix (position on the deletion/contraction tree) up to (but not including) the last contraction. This function then makes that step a deletion. The function also checks to see whether the last matrix was obtained entirely from deletions, in which case it signals that the computation has finished by returning the number 0.

Once the computation has finished, the function then prints the Tutte polynomial and signals to the computer that it is done by returning the number 0.

3.5.2 Basic Matrix Functions

`**imatrix` and `*ivector`

These two programs are incorporated with minor modifications from the book *Numerical Recipes in C*. They allocate memory for a dynamically sized matrix or vector, respectively. This is necessary because the program doesn't know how large the user's matrix is when it is compiled. For `**imatrix` The starting entry for the matrix is set by the variable `start`. Thus, inputing `start=2` will have the matrix begin with the entry $a_{2,2}$. Similarly `*ivector` will create a matrix with entries indexed by numbers running from `low` to `high`.

`initpoly`

This function was written to initialize each entry of the Tutte matrix to 0. I'm sure there are better ways to accomplish this, but it ends up saving some trouble later.

`printimatrix`, `printvector`, and `printpoly`

These are two straightforward functions to print matrix-indexed values in either matrix or polynomial form. `printimatrix` prints the entries of an integer matrix row by row, with square brackets at the ends of each row. `printvector` does the same, but prints only one row (this function is a vestige of the troubleshooting stage of writing the program, but I left it in just in case someone needs it). The `printpoly` function employs three tests in order to generate a nicer looking output. First, it tests the coefficient of a term, only printing if it is non-zero. Second, it tests whether it is the first term of the polynomial. This is the easiest way to print the polynomial without either beginning or ending with a "`+` "sign. Lastly, it only prints a power of `x` or `y` if it is non-zero.

If you need a specialized output for use in another application, these are the functions to modify.

`nrerror`

This function is verbatim from *Numerical Recipes in C*. It is used to handle errors in the memory allocation functions without crashing the computer. Edit at your own risk.

`getcols`

This function traverses a matrix allocated with `**imatrix` column by column and assigns to it integer values read from a file. Integers are read as per standard written English: left to right, top to bottom. I'm not entirely sure how it works, but it'll behave nicely if you follow the instructions at the beginning of the manual. The temporary variable `temp` isn't necessary, but it saves some pain in trying to figure out what pointers go where. (For the uninitiated, pointers are objects with which `C` programmers are intimately familiar and are the reason for all the `*`s in the code.)

`freeimatrix`

This function was another of the ones taken from *Numerical Recipes in C*. It is actually never used in the program, although it was used in several earlier, less efficient, versions.

`copyimatrix`

Also used more often in earlier incarnations of the code (it is used only once in this one), `copyimatrix` will overwrite values of one integer matrix (`m`) onto another (`newmat`).

3.5.3 Row-reduction Functions

`swaprows` and `swapcols`

These two functions, respectively, exchange two rows or columns of a matrix. They go entry by entry along the rows or columns and use the following standard swapping technique. To achieve

$a \leftrightarrow b$, create a temporary variable c (in the code, `temp`) and perform the assignments: $a \to c$, $b \to a$, then $c \to b$.

`setuppivot`

As the name implies, this function is responsible for putting a non-zero value in a given pivot position, the (`pivrow`,`pivcol`) entry of the matrix. To be fair, it doesn't quite do this. As you'll see in the `rowreduce` function, it is actually sufficient for this program to try its best to move a non-zero entry into the pivot position without switching any columns. It does so by systematically swapping rows into the pivot row from below until it either finds a valid entry or runs out of rows. The `temp` variable is responsible for incrementing the row that gets swapped, preventing an infinite loop of swapping and rejecting the same two rows.

`elim`

Once a pivot is located, the `elim` function uses integer row operations to make all other entries in the pivot column zero. The two sections of the function are identical, except that the first deals with the lower rows of the matrix and the second deals with the upper rows. Both work by going row by row and then entry by entry, changing the entries through a combination of scaling and subtraction. If the pivot column has an entry a_0 in the pivot row and b_0 in the other row, then, denoting entries from the pivot and non-pivot row respectively a and b, the function makes the following substitution:

$$b \to a_0 b - b_0 a$$

as it traverses the non-pivot row column by column. Note that this will send $b_0 \to 0$, as desired, without violating any rules regarding linear row combination.

`rowreduce`

`rowreduce` is the function that, with the help of the preceeding functions, actually row-reduces a given matrix. It starts in the upper left corner of the matrix and calls the `setuppivot` function. If `setuppivot` is successful, indicated by a non-zero entry in the desired position, `rowreduce` calls the `elim` function to eliminate other entries in the current column. Since the `elim` function involves a great deal of scaling, `rowreduce` also checks if any entry of the matrix exceeds the `TH` value (see The Lowest Terms Threshold section), and attempts to rescale rows of the matrix if that is the case. It then moves diagonally down and to the right ny one row and one column and starts again. If `setuppivot` fails to find a non-zero entry, the `rowreduce` function simply moves to the next column and continues searching. This process repeats until the matrix runs out of rows or columns, at which point the matrix will be completely row-reduced.

3.5.4 Factoring

`lowestterms`, `killfactor`, and `iabs`

This function divides out common factors from the Prime List in each row of a matrix. Going row by row, it first tests for a row of zeros (which, as you might imagine, has many common factors). The function uses an indicator variable `test` to indicate when a no more factors of a given number appear. `test`=0 if each entry in the row is divisible by the current number. If there is a row of zeros, `test` is preemptively assigned the value 1.

`lowestterms` then goes through the Prime List and calls `killfactor` for each prime. `killfactor` takes a given value `factor` and uses it as a modular base. It then goes through each entry in the given matrix row, takes its absolute value (using `iabs`), and finds its value mod `factor`. `test` is then incremented by that value. If every term is divisible by `factor`, then `test` remains at zero. Otherwise, it attains a non-zero value. If `test` equals zero, each entry in the row is divided by `factor` and the process is repeated until `factor` is no longer a common factor to the row.

`iabs` is short for *integer absolute value*. It returns the input value if it is greater than or equal to zero and -1 times the value otherwise.

`toobig`

This function goes through each entry of the matrix and finds the maximum value. If this value exceeds the preset threshold (`TH`), the function returns a 1. Otherwise, it returns a 0.

3.5.5 Tutte Functions

`justloopsnbridges`

This function is used to indicate when a base case has been reached in the algorithm. As the name suggests, the bulk of the function is involved in searching the matroid for edges which are neither loops nor bridges. First, it checks for the base case of a matroid in one dimension. These are easy matroids for which to find Tutte, and are treated separately in order to save time. If such a base case is found, the function returns the number -1. Otherwise, it searches for loops and bridges by going row by row and tallying the number of non-zero entries. If this tally, stored in the variable `count`, is greater than one, the `indicator` variable is incremented, indicating that there is a non-isthmus, non-loop vector with a non-zero component in the current row. The second `if` loop searches down that row, this time using `count` as an indicator variable, and finds the first edge with a non-zero component and returns its index in the matrix. This information is used by the `main` function to decide which edge to delete or contract during the algorithm. If no row is found with more than one non-zero entry it means that, for a row-reduced matrix, every vector is either a loop (all zeros) or a bridge (linearly independent). To indicate that such a situation has arisen, the function returns a zero.

`done`, `lastreczero`, and `lastrecmod`

These three functions search the `record` vector, which keeps track of deletions and contractions. Recall that the program's algorithm traverses the partial binary tree of deletions and contractions in the Tutte computation in a depth-first manner. This means that it will contract the matroid until it hits a base case. Then, it will repeat the construction up until the last contraction and make that contraction a loop, then continue contracting until it reaches a base case. Contractions are stored as 1s and deletions as 0s, starting at the last entry of the `record` vector. `lastreczero` is used for determining where to store the next entry in the `record` vector. The function returns the index of the vector's last unused slot. `lastrecmod` returns the index of `record`'s most recent contraction. It saves an insubstantial amount of run-time by using a combination assignment and the built-in remainder function instead of directly testing each entry for equality to 1. Anyone familiar with the expression "penny wise but pound foolish" surely appreciates my programming approach. `done` uses the remainder/assignment functions a bit more cleverly, and returns a zero (indicating that the program and its depth-first search are done) if no odd numbers are present in the `record` vector.

`sendtotutte` and `d1totutte`

Once the program reaches a base case, it must convert it into a term or set of terms in $T_M(x, y)$. For the base case of all loops and isthmuses, the program uses `sendtotutte`. The `sendtotutte` function simply goes column by column and looks for a non-zero entry. If it finds one, it increments the power of x (`a` in the code). Otherwise, it increments the power of y (`b` in the code). This function also treats the cases of zero rows or zero columns separately in order to avoid algorithmic confusion.

`d1totutte` treats the base case of a matroid in one dimension. If such a matroid has n vectors, i of which are non-zero, it can be shown by induction that its Tutte polynomial is equal to (for $n, i > 0$)

$$xy^{n-i} + \sum_{j=n-i+1}^{n-1} y^j$$

Accordingly the function counts the number of non-zero vectors and increments the appropriate terms of the Tutte polynomial.

`deletein` and `contractin`

These two functions perform the deletion and contraction operations, given an input edge, for the program. `deletein` moves every vector to the right of the deleted edge over by one column and then tells the computer to ignore the last column by decrementing the dimension variable `n`. `contractin` searches the given edge for its non-zero entry. It then moves entries around to the effect of moving everything below the located row up by one row and everything to the right of the given edge to the left by one column. The function also decrements the `d` and `n` dimension variables to effectively shrink the matrix. To elaborate slightly, all functions in the program operate on a matrix based on inputs of the number of rows and columns and an input locating the values of the matrix itself.

Thus, if the function is given a value for `d` (number of rows) or `n` number of columns that is smaller than the actual size of the matrix, it only operates on the first `d` rows and first `n` columns. This small observation, it turns out, saves oodles of computation time and power because it allows one to "resize "the matrix without actually resizing it.

`rebuild`

This function deals with a matrix after it has been processed from a base case. It begins by restoring it to its original, user-inputed, values and then follows the record of deletions and contractions as in the `main` function up until (but not including) the last contraction in the record. It then changes this to a deletion, checks to see if the program is done (using the `done` function) and updates the record. The function, when finished, returns the value of the `done` function, telling the `main` function whether or not to continue operating.

3.6 Making it Mod P

`tutte_modp.c` has most of the same functions as `tutte.c`. The main difference is that instead of performing all the operations related to factoring it simply finds the number modulo the given factor `p`. It does so using the `mod` function. `mod` takes every entry in the matrix and uses `C`'s built-in remainder function to find its value mod p.

Chapter 4

Feedback

All praise should be directed toward my advisor, Professor Victor Reiner, at reiner@math.umn.edu. Any comments, bugs, or suggestions should be sent to me, Michael Barany, at bara0051@umn.edu. General complaints may be directed to someone_else@yahoo.com. Vic and I are, of course, curious about how people are using this program, so please e-mail either of us and tell us what you're up to. If it's not 2006 yet, we're probably still eager for anecdotal evidence of how the program runs with different matroids.

Chapter 5

Acknowledgments

This work was supported by the University of Minnesota Undergraduate Research Opportunities Program. I would like to thank my advisor Vic and Dr. Jeremy Martin for introducing me to $T_M(x, y)$ and directing me toward Vic. Alex Miller held my hand as I took my first baby steps into `C` programming has been helpful throughout this project. The U of M Math Department computing staff is fantastic as well.