TUTTEPOL: A well-performing C++ implementation of an efficient recursive algorithm for computing Tutte polynomials

Henrik Kragh Sørensen

April 26, 2007

Abstract

This is the documentation for the C++ implementation TUTTEPOL of a recursive algorithm for efficiently computing Tutte polynomials. The algorithm was developed by Thomas Britz in 2006, see (Britz, Britz, Shiromoto and Sørensen [accepted]). It has been implemented using Maple by Thomas Britz, using Fortran by Dieter Britz and using C++ by Henrik Kragh Sørensen.

Contents

1 Introduction and overview

The TUTTEPOL implementation was programmed using Microsoft Visual C++ but done in a way that preserves portability. Thus, it has been unproblematically ported to Linux using g++ (see appendix F for makefile and homepage for instructions).

The implementation uses some object oriented techniques, but only to a limited extent such as helping allocating and de-allocating data for the recursive stack and keeping parameters together (which could also have been achieved using structs.

`TuttePol.exe` calculates the Tutte polynomial of a binary matrix A. The matrix A is assumed to be of the form $A = [I|D]$, and only the submatrix D is given as input to `TuttePol.exe`.

Running as a command line programme, `TuttePol.exe` is called as

```
TuttePol <filename>
```

where `<filename>.txt` contains the submatrix D in the following format:

line 1 contains the dimensions of D, separated by a space, and

lines 2– contain D with 0s and 1s separated by spaces.

Upon completion, the output is written to the file `<filename>.polynomial` with a computation log file written to `<filename>.log`. During computation, dots will appear on the screen for each garbage collection to show that the program is still running (some of the interesting computations will take more than a few seconds!)

2 Implementation: Outline

The main hook for TUTTEPOL is in the file `TuttePol.cpp` (appendix A), where the `main()` function is implemented. It allocates structures, parses arguments, loads the matrix, initiates the computation, writes the output, and exits.

The input matrix file has the following format: The first line contains the number of rows r and number of columns k seperated by a space. Then follows r lines of k symbols (zero or one) seperated by commaes or spaces.

The central part of the computation is implemented in the member function called `ComputeTuttePol()` of the `Matrix` class (see appendix B). This function performs the step of the algorithm, calling itself recursively as well as relying on other supportive functions. The `ComputeTuttePol` function is documented in more detail in Section 3.1.

The resulting polynomials are implemented using the class `Polynomial` (see appendix C). Polynomials are constructed as either terms (of the form $cx^n y^m$) or sums of such terms. Upon simplification, sums of terms can be collapsed into more effiently handled structures, called “multiterms”. The data structures of polynomials are documented in Section 3.3.

For optimization, polynomials corresponding to "small" matrices are kept in a cache implemented in the class `PolyStash` (see appendix D). The functioning and implementation of this cache is documented in Section 3.5.

TUTTEPOL relies on 64-bit words which are defined in `StdAfx.h` (see appendix E).

3 Implementation: Features

In this section, key aspects of the implementation are documented.

3.1 Implementing the algorithm

`ComputeTuttePol`. The core of the recursive algorithm is implemented in the function `Matrix->ComputeTuttePol()` (`Matrix.cpp`, line 33). This function computes the Tutte polynomial of its matrix and returns it as a `Polynomial` object. It makes use of supportive routines for its lower-level data handling.

The bases of the recursive function are tested at the head of the function. These are:

- Matrix has only one row ($r = 1$),
- Matrix has only one column ($k = 1$),
- Matrix is eligible for the stash and has previously been computed (see Section 3.5).

If the function has not returned as the result of one of these conditions, the computation of the Tutte polynomial continues according to the algorithm.

First, the first non-zero entry in the first row (the pivot) is found. The index of the corresponding column is kept in the variable j.

If either the entire first row is a zero row ($j = k + 1$, the first column of A is a co-loop) or the pivot occurs in the last column ($j = k$, that column is a co-loop in $A \setminus 1$), the computation of the Tutte polynomial is effected recursively using the function `ComputePluckMatrix` (see below).

In the contrary case, if the entire first row to the right of the pivot consists of zero entries (the j'th column is a co-loop of $A \setminus 1$), the Tutte polynomial can also be computed recursively using `ComputePluckMatrix`.

Left with the most general case, the row reduction is effected, and the function calls itself, recursively.

`ComputePluckMatrix`. The key functionality of removing rows or columns from a matrix is implemented using the function `ComputePluckMatrix` (`Matrix.cpp`, line 295).

The function is called to fill a newly created `Matrix` object based on a given one (often the present one, **`this`**). Apart from the reference to a `Matrix` object, it takes two parameters r and c. These both have the following meaning: If $r = 0$, no rows are removed; otherwise, the row with the zero-index $r - 1$ is removed. Similarly for columns.

3.2 Data representation of Matrix

Matrix entries. The `Matrix` object has entries that are binary. To conserve memory and optimize computation, these binary entries have been packed into 8-bit words. Therefore, the functions to fetch and store individual entries (`Matrix->GetElement` and `Matrix->SetElement`, respectively) rely on the `ComputeDataIndex` function to compute the correct index.

Row reduction. Optimization has been possible by using `memcpy` for memory copying of larger regions of entries rather than individual calls to `SetElement` and `GetElement`.

Furthermore, the operation of the function `RowReduce` (`Matrix.cpp`, line 168) has been optimized by relying on the logical operation of "exclusive-or" (`a^b` in C++ notation) collapsing 8 binary operations into one.

3.3 Data representation of Polynomial

The polynomials are basically represented as linked lists of nodes. Each node can either by a sum of other nodes or an atom in the form of a term ($cx^n y^m$) or a "multiterm".

The multiterms are collected sums of atomic terms. The multiterms are, themselves, atoms. Multiterms are represented by a polynomium-matrix that contains the coefficients c_{ij} such that i and j are bounded to include only the "rectangle" of the polynomium-matrix that has non-zero entries.

Linked lists of terms and multiterms are collapsed into a multiterm using the functions `Polynomial->Simplify` and `Polynomial->UpdateCoefficients`. The latter function computes the above-mentioned polynomium-matrix, whereas the former function copies the span of non-zero entries into a new multiterm node.

3.4 Garbage collection in Polynomial

Simplification of polynomials is called in the process of garbage collection. This approach allows for the fast (and unscrupulous) allocation of `Polynomial` objects. Whenever the height of new objects (the number of nodes below it in the linked list) exceeds a fixed constant, the terms are collapsed into a multiterm structure.

The present settings allows for a powerful execution with only very limited memory usage.

3.5 The Stash

The cache was implemented to speed up computation of Tutte polynomials near the bottom of the recursive execution stack.

Polynomials corresponding to matrices with $r + k \leq 7$ and $r, k > 1$ are cached. A 16-bit key is computed in `Matrix->ComputeStashIndex` (`Matrix.cpp`, line 336) and used to reference the polynomial in the cache. The packing is as follows:
bit 0–1 : $r - 2$
bit 2–3 : $k - 2$
bit 4–15 : matrix entries
However, this key computation only utilizes some 17% of the key space. A more efficient utilisation of the key space has to be balanced against the ease of computing the key.

4 A few considerations of efficiency and results

A potential mod p **version.** The modularisation of data handling into the supportive routines of the `Matrix` class makes it relatively simple to implement the algorithm for $\mathbb{Z}/_{p\mathbb{Z}}$ rather than the binary case $\mathbb{Z}/_{2\mathbb{Z}}$ implemented in this version. The only changes have to be done in the supportive routines (data allocation, data handling, and row reduction) and in testing for zero entries in the first row. However, this is an added functionality to be implemented in a subsequent and different version because the required testing is likely to slow the computation in the binary case.

Trade-offs. The present implementation offers good efficiency. The trade-offs made and already mentioned include:

- Data representation as 8-bit words prohibits an integrated mod$p$ version because of numerous tests for global variable.
- Caching polynomials of small matrices (needs frequent requests to be efficient).
- Ease of computing the cache key versus the utilisation of the key space.

The program was used to compute both small and large examples. Its use of 64-bit words for the coefficients of the polynomials is due to the larger examples that it was intended for.

Bibliography

Britz, D., Britz, T., Shiromoto, K. and Sørensen, H. K.: [accepted], The higher weight enumerators of the doubly-even, self-dual [48, 24, 12] code. Accepted for publication in *IEEE Transactions of Information Theory* 2007/03/27. Expected size: 5 pages. To appear 2007.

A Main code (entry point)

```
// $Header: /home/cvs/cvsroot/MSVC/TuttePol/TuttePol.cpp,v 1.26 2006/12/20 23:14:21 hkragh Exp $
// TuttePol.cpp : Defines the entry point for the console application.

#include "StdAfx.h"

#include <stdlib.h>
#include <stdio.h>
#include <time.h>

#include "Matrix.h"
#include "Polynomial.h"

uint64 fourbyfour=0;
uint64 dimmaxseven=0;
uint64 dimmaxeight=0;
uint64 dimeight=0;

#include "PolyStash.h"
PolyStash* ps;

#include "VersionInfo.cpp"

char global_maxx;
char global_maxy;

int main(int argc, char* argv[])
{
    fprintf(stdout, "TuttePol (hkragh) version %s%s, built %s\n", MAJORVERSION,
        MINORVERSION, __DATE__);

    if (argc!=2) {
        fprintf(stderr, "Syntax: %s <matrix name>\n", argv[0]);
        return -1;
    }

    if (getenv("HOSTNAME") != NULL) {
        fprintf(stdout, "Running on computer %s\n", getenv("HOSTNAME"));
    } else if (getenv("COMPUTERNAME") != NULL) {
        fprintf(stdout, "Running on computer %s\n", getenv("COMPUTERNAME"));
    }

    fprintf(stdout, "Operating on matrix %s\n", argv[1]);

    char matrixfilename[255];
    char polynomialfilename[255];
    sprintf(matrixfilename, "%s.txt", argv[1]);
    sprintf(polynomialfilename, "%s.polynomial", argv[1]);

    fprintf(stdout, "Matrix input file %s\n", matrixfilename);
    fprintf(stdout, "Polynomial output file %s\n", polynomialfilename);

    Matrix* mr = new Matrix;
    if (mr->ReadFromFile(matrixfilename)) {
        mr->Print(stdout);

        ps = new PolyStash();
```

```
        time_t t0 = time(NULL);
        Matrix* m;
        if (0) {
            m = new Matrix();
            m->ComputePluckMatrix(mr,1,0);
            delete mr;
        } else {
            m=mr;
        }

        time_t t1=t0;
        fprintf(stdout, "Main computation started at %s", asctime(localtime(&t1)));
        Polynomial* p = m->ComputeTuttePol();
        time_t t2 = time(NULL);

        fprintf(stdout, "\n");
        fprintf(stdout, "Main computation ended   at %s", asctime(localtime(&t2)));
        fprintf(stdout, "Time to compute %d sec\n", t2-t1);

        if (p == NULL) {
            fprintf(stderr, "Error computing Tutte polynomial\n");
            return -1;
        }
        Polynomial* q = p->Simplify();
        time_t t3 = time(NULL);
        fprintf(stdout, "Time to reduce  %d sec\n", t3-t2);
        q->Print(polynomialfilename);
        q->Print(stdout);
        fprintf(stdout,"\n");
        time_t t4 = time(NULL);
        fprintf(stdout, "Time to print   %d sec\n", t4-t3);

        delete p;
        delete q;
        time_t t5 = time(NULL);
        fprintf(stdout, "Time to kill    %d sec\n", t5-t4);

        fprintf(stdout, "Total run time  %d sec\n", t5-t0);

        float f = 100*(float)ps->m_actives / STASHSIZE;
        fprintf(stdout, "\nTotal stash usage at %2.1f%% (%ld)\n", f, ps->m_actives)
            ;
    }

    delete ps;
    return 0;
}
```

B The Matrix class

```
// $Header: /home/cvs/cvsroot/MSVC/TuttePol/Matrix.cpp,v 1.22 2006/12/21 00:42:14 hkragh Exp $

#include "StdAfx.h"
#include "Matrix.h"

#include <string.h>
#include <stdlib.h>

unsigned char BYTEINDEX[8] = { 1, 2, 4, 8, 16, 32, 64, 128};
unsigned char BYTEINDEXACC[8] = { 0, 1, 3, 7, 15, 31, 63, 127};

extern char global_maxx;
extern char global_maxy;

#include "PolyStash.h"
extern PolyStash* ps;


Matrix::Matrix(void)
{
    m_data = NULL;
    m_rows = 0;
    m_columns = 0;
}

Matrix::~Matrix(void)
{
    if (m_data != NULL) {
        free(m_data);
    }
}

Polynomial* Matrix::ComputeTuttePol()
{
    COLROWTYPE r = this->GetDimensionRows();
    COLROWTYPE k = this->GetDimensionColumns();

    if (r==1) {
        int nl = 0;
        for (COLROWTYPE j=1; j<=k; j++) {
            if (this->GetElement(j-1,0)==0) { nl++; }
        }
        Polynomial* p = new Polynomial(1,1,nl); // x*y^nl
        for (COLROWTYPE i=nl+1; i<=r+k-1; i++) {
            p = new Polynomial(POLY_SUM, p, new Polynomial(1,0,i));
                // y^i  for  i = nl+1,...,r+k-1
        }
        return p; // x*y^nl + y^(nl+1) + ... + y^(r+k-1)
    }

    if (k==1) {
        int ncl=0;
        for (COLROWTYPE j=1; j<=r; j++) {
            if (this->GetElement(0,j-1)==0) { ncl++; }
        }
        Polynomial* p = new Polynomial(1,ncl,1);    // x^ncl * y
```

```
        for (COLROWTYPE i=ncl+1; i<=r+k-1; i++) {
            p = new Polynomial(POLY_SUM, p, new Polynomial(1,i,0));
                // x^i for i = ncl+1,...,r+k-1
        }
        return p;
    }

    STASHINDEX key = this->ComputeStashIndex(1);

    if ( ps->CheckStash(key) ) {
        return ps->GetPoly(key);
    }

    COLROWTYPE j=1;
    while ((j<=k) && (this->GetElement(j-1,0)==0)) { j++; }

    if (j==k+1) {
        Matrix B;
        B.ComputePluckMatrix(this,1,0); // Remove first row
        // x * TuttePol(B)
        Polynomial* p = B.ComputeTuttePol();
        p->MultiplyByX();
        return ps->SetStash(key, p);
    }

    if (j==k) {
        Matrix B;
        B.ComputePluckMatrix(this,1,0); // Remove first row
        Matrix C;
        C.ComputePluckMatrix(this,1,k); // Remove first row and k'th (=j'th) column
        Polynomial* p = C.ComputeTuttePol();
        p->MultiplyByX();
        return ps->SetStash(key, new Polynomial(POLY_SUM, B.ComputeTuttePol(), p));
    }


    bool nc=true;
    for (COLROWTYPE jj=j+1; jj<=k; jj++) {
        if (this->GetElement(jj-1,0)!=0) { nc=false; }
    }

    Matrix B;
    B.ComputePluckMatrix(this,1,0);     // Remove first row

    if ( nc ) {
        Matrix C;
        C.ComputePluckMatrix(this,1,j); // Remove first row and j'th column
        Polynomial* p = C.ComputeTuttePol();
        p->MultiplyByX();
        return ps->SetStash(key, new Polynomial(POLY_SUM, B.ComputeTuttePol(), p));
    }


    // Row reduction
    for (COLROWTYPE ii=2; ii<=r; ii++) {
        if (this->GetElement(j-1,ii-1)) {
            this->RowReduce(ii-1,0,j-1);
        }
    }
```

```
    Matrix C;
    C.ComputePluckMatrix(this,0,j); // Remove j'th column
    C.RowRoll(1);                   // and rotate rows one down

    return ps->SetStash(key, new Polynomial(POLY_SUM, B.ComputeTuttePol(), C.
        ComputeTuttePol()));
}

DATATYPE Matrix::GetElement(COLROWTYPE c, COLROWTYPE r)
{
    // Lean-and-mean (ie no bounds-checks)
    return ((m_data[ComputeDataIndex(c,r)] & BYTEINDEX[c%8]) == BYTEINDEX[c%8]);
}

void Matrix::SetElement(COLROWTYPE c, COLROWTYPE r, DATATYPE val)
{
    // Lean-and-mean (ie no bounds-checks)
    if (val) {
        m_data[ComputeDataIndex(c,r)] |= BYTEINDEX[c%8];
    }
    else {
        m_data[ComputeDataIndex(c,r)] &= ~BYTEINDEX[c%8];
    }
}

unsigned int Matrix::ComputeDataIndex(COLROWTYPE c, COLROWTYPE r)
{
    return r*(m_columns/8+1)+(c/8);
}


COLROWTYPE Matrix::GetDimensionRows()
{
    return m_rows;
}

COLROWTYPE Matrix::GetDimensionColumns()
{
    return m_columns;
}

bool Matrix::Print(FILE* fp)
{
    for (COLROWTYPE r=0; r<this->GetDimensionRows(); r++) {
        for (COLROWTYPE c=0; c<this->GetDimensionColumns(); c++) {
            fprintf(fp, "%d ", this->GetElement(c,r));
        }
        fprintf(fp, "\n");
    }
    return true;
}

bool Matrix::RowReduce(COLROWTYPE r1, COLROWTYPE r2, COLROWTYPE j)
{
    // Reduce row r1 by row r2
    // r1, r2: zero-based indices of rows to reduce
    // j: zero-based index of first column in reduction (default: 0)
    unsigned int datapos1 = this->ComputeDataIndex(j,r1);
```

```
    unsigned int datapos1a = this->ComputeDataIndex(this->GetDimensionColumns(),r1)
        ;
    unsigned int datapos2 = this->ComputeDataIndex(j,r2);

    // Special first byte
    unsigned char mask = BYTEINDEXACC[j%8];
    m_data[datapos1] = (m_data[datapos1] & mask) | ((m_data[datapos1]^m_data[
        datapos2]) & ~mask);

    // Regular bytes
    for (datapos1++;datapos1<=datapos1a; datapos1++) {
        datapos2++;
        m_data[datapos1] ^= m_data[datapos2];
    }
    return true;
}

bool Matrix::RowSwap(COLROWTYPE r1, COLROWTYPE r2)
{
    for (COLROWTYPE c=0; c<this->GetDimensionColumns(); c++) {
        DATATYPE t = this->GetElement(c,r1);
        this->SetElement(c,r1,this->GetElement(c,r2));
        this->SetElement(c,r2,t);
    }
    return true;
}

bool Matrix::RowRoll(COLROWTYPE n)
{
    if (n<1) { return false; }
    if (n>=m_rows) { return false; }

    DATATYPE* temp = (DATATYPE*)malloc(n*sizeof(DATATYPE));
    COLROWTYPE r,c;
    for (c=0; c<m_columns; c++) {
        for (r=0; r<m_rows; r++) {
            if (r<n) {
                temp[r] = GetElement(c,r);
            } else {
                SetElement(c,r-n,GetElement(c,r));
            }
        }
        for (r=0; r<n; r++) {
            if (m_rows-n+r<0) {
                fprintf (stderr, "Overflow (m_rows,n,r)=(%d,%d,%d)\n", m_rows, n, r
                    );
            }
            SetElement(c,m_rows-n+r,temp[r]);
        }
    }
    free(temp);
    return true;
}

bool Matrix::ReadFromFile(char* filename)
{
    fprintf(stdout, "Reading matrix from file %s...\n", filename);
    FILE* mf = fopen(filename, "r");
    if ( mf == NULL ) {
```

```
230         fprintf(stderr, "Error opening file %s\n", filename);
            return false;
        }

        int rows = 0;
235     int columns = -1;

        char* line = (char*)malloc(1024*sizeof(char));
        // read specification line
        fgets(line,1024,mf);
240     // parse specification line
        if (sscanf (line, "%d %d", &rows, &columns) != 2) {
            return false;
        }

245     if (columns<0) {
            fprintf(stderr, "Malformed line (%d) in matrix description\n", -columns);
            free(line);
            return false;
        }
250
        fprintf(stdout, "Reading %d rows of %d columns from file %s\n", rows, columns,
            filename);

        if (m_data != NULL) { free(m_data); }

255     m_rows = rows;
        m_columns = columns;
        this->AllocateData(m_columns, m_rows);

        int row = 0;
260     while (fgets(line, 1024, mf)!=NULL) {
            int column=0;
            for (unsigned int c=0; c<strlen(line); c++) {
                unsigned int c1;
                for (c1=c+1; column<columns && c1<strlen(line) && line[c1]!=',' && line
                    [c1]!=' '; c1++) {};
265             // Column data is in line[c..c1[
                SetElement(column,row,atoi(line+c)==1);
                column++;
                c=c1;
            }
270         row++;
        }

        free(line);
        fclose(mf);
275
        // NB: Powers of x are bounded by number of rows; powers of y by number of
            columns
        global_maxx = this->GetDimensionRows();
        global_maxy = this->GetDimensionColumns();

280     return true;
    }

    bool Matrix::AllocateData(COLROWTYPE columns, COLROWTYPE rows)
    {
285     unsigned int d = ((columns/8)+1)*rows;
```

```
    if (m_data != NULL) {
        free(m_data);
        fprintf(stderr,"Error in data allocation\n");
        exit(1);
    }
    m_data = (unsigned char*)calloc(d,sizeof(unsigned char));
    return true;
}

void Matrix::ComputePluckMatrix(Matrix *m, COLROWTYPE r, COLROWTYPE c)
{
// if r=0: no action on rows; r>0: pluck row r-1
// if c=0: no action on columns; c>0: pluck column c-1
    COLROWTYPE br = m->GetDimensionRows();
    COLROWTYPE bn = m->GetDimensionColumns();
    m_rows = br;
    if (r>0) {
        m_rows--;
    }
    m_columns = bn;
    if (c>0) {
        m_columns--;
    }
    this->AllocateData(m_columns, m_rows);

    if ( (c==0) && (r==1) ) { // No change on columns, remove first row (typical
        use)
        unsigned int datapos1a = m->ComputeDataIndex(0,1);
        unsigned int datapos1b = m->ComputeDataIndex(bn-1,br-1);
        unsigned int datapos2a = this->ComputeDataIndex(0,0);
        memcpy((void*)(this->m_data+datapos2a*sizeof(unsigned char)),
            (void*)(m->m_data+datapos1a*sizeof(unsigned char)),
            (datapos1b-datapos1a+1)*sizeof(unsigned char));
        return;
    }

    int bi = 0;
    for (COLROWTYPE i=0; i<br; i++) {
        if (i+1!=r) {
            int bj = 0;
            for (COLROWTYPE j=0; j<bn; j++) {
                if (j+1!=c) {
                    this->SetElement(bj,bi, m->GetElement(j,i));
                    bj++;
                }
            }
            bi++;
        }
    }
}

unsigned long Matrix::ComputeStashIndex(int type)
{
    if (type==1) {
        if ( m_columns+m_rows <= 7 ) {
            unsigned long res;
            res = (m_rows-2) + (m_columns-2)*4;

            unsigned long factor = 16;
```

```
                for (COLROWTYPE i=0; i<m_rows; i++) {
345                 res += (m_data[i] & BYTEINDEXACC[m_columns]) * factor;
                    factor *= BYTEINDEX[m_columns];
                }
                return res;
            }
350     }
        return 0;
    }
```

C The Polynomial class

```
// $Header: /home/cvs/cvsroot/MSVC/TuttePol/Polynomial.cpp,v 1.19 2006/12/21 00:42:14 hkragh Exp $

#include "StdAfx.h"
#include "Polynomial.h"
#include <stdlib.h>
#include <string.h>         // Required for memcpy
#include <stdio.h>

#define GB_TRIGGER 25

extern char global_maxx;
extern char global_maxy;

bool PrintTerm(FILE* fp, uint64 coef, int xpower, int ypower)
{
    if (coef==0) { return false; }
    else if (coef==1) { }
    else { fprintf(fp, "%I64Lu", coef); }

    if (xpower==0) { }
    else if (xpower==1) { fprintf(fp, "x"); }
    else if (xpower<10) { fprintf(fp, "x^%d", xpower); }
    else { fprintf(fp, "x^{%d}", xpower); }

    if (ypower==0) { }
    else if (ypower==1) { fprintf(fp, "y"); }
    else if (ypower<10) { fprintf(fp, "y^%d", ypower); }
    else { fprintf(fp, "y^{%d}", ypower); }
    return true;
}

bool GBC(int height) {
    if (0) {
        return (height % GB_TRIGGER == GB_TRIGGER-1);
    } else {
        return (height>=20);
    }
}

Polynomial::Polynomial(POLY_TYPES type, Polynomial* p1, Polynomial* p2)
{
    m_type=type;
    m_coefs=NULL;

    if (GBC(p1->m_height)) {
        fprintf(stderr, ".");
        Polynomial* q = p1->Simplify();
        delete p1;
        p1 = q;
    }
    if (GBC(p2->m_height)) {
        fprintf(stderr, ".");
        Polynomial* q = p2->Simplify();
        delete p2;
        p2 = q;
    }
```

```
    m_p1 = p1;
    m_p2 = p2;
    SetHeight();
}

Polynomial::Polynomial(POLY_TYPES type, uint64 coef, int xpower, int ypower)
{
    m_type=POLY_TERM;
    m_p1 = NULL;
    m_p2 = NULL;
    m_coef = coef;
    m_coefs=NULL;
    m_xpower = xpower;
    m_ypower = ypower;
    m_height=0;
}

Polynomial::Polynomial(uint64 coef, int xpower, int ypower)
{
    m_type=POLY_TERM;
    m_p1 = NULL;
    m_p2 = NULL;
    m_coef = coef;
    m_coefs=NULL;
    m_xpower = xpower;
    m_ypower = ypower;
    m_height=0;
}

Polynomial::Polynomial(uint64* coefs, int minx, int miny, int maxx, int maxy, int
    xpower)
{
    m_type = POLY_MULTITERM;
    m_coefs = coefs;
    m_p1 = NULL;
    m_p2 = NULL;
    m_coef = 0;
    m_xpower = xpower;
    m_ypower = 0;
    m_height=0;
    m_minx = minx;
    m_miny = miny;
    m_maxx = maxx;
    m_maxy = maxy;
}

Polynomial::~Polynomial(void)
{
    if (m_p1) { delete m_p1; m_p1 = NULL; }
    if (m_p2) { delete m_p2; m_p2 = NULL; }
    if (m_coefs != NULL) { free(m_coefs); m_coefs=NULL; }
}

bool Polynomial::Print(char* filename)
{
    FILE* fp = fopen(filename, "w");
    if (fp == NULL) {
        fprintf(stderr, "Error opening file %s\n", filename);
```

```
115         return false;
        }
        Print(fp);
        fclose(fp);
        return true;
120 }

    bool Polynomial::Print(FILE* fp)
    {
        if (m_type == POLY_SUM) {
125         m_p1->Print(fp);
            fprintf(fp, "\n+");
            m_p2->Print(fp);
        } else if (m_type == POLY_TERM) {
            return PrintTerm(fp, m_coef, m_xpower, m_ypower);
130     } else if (m_type == POLY_MULTITERM) {
            bool notfirst = false;
            for (int x=m_maxx; x>=m_minx; x--) {
                for (int y=m_maxy; y>=m_miny; y--) {
                    uint64 coef = this->GetCoefficient(x,y);
135                 if ((coef!=0) && (notfirst)) {
                        fprintf (fp, "\n+");
                    }
                    notfirst = notfirst | PrintTerm(fp, coef, x, y);
                }
140         }
        }
        return true;
    }

145 int update_maxx;
    int update_maxy;
    int update_minx;
    int update_miny;

150 Polynomial* Polynomial::Simplify()
    {
        uint64* coefs = (uint64*)calloc((global_maxx+1)*(global_maxy+1),sizeof(uint64))
            ;
        update_maxx = update_maxy = 0;
        update_minx = global_maxx+1;
155     update_miny = global_maxy+1;

        UpdateCoefficients(coefs);

        uint64* ac = (uint64*)calloc((update_maxx-update_minx+1)*(update_maxy-
            update_miny+1),sizeof(uint64));
160     for (int x=update_minx; x<=update_maxx; x++) {
            for (int y=update_miny; y<=update_maxy; y++) {
                ac[(x-update_minx)*(update_maxy-update_miny+1)+(y-update_miny)] = coefs
                    [x*(global_maxy+1)+y];
            }
        }
165
        free(coefs);
        coefs=NULL;
        return new Polynomial(ac, update_minx, update_miny, update_maxx, update_maxy,
            0);
    }
```

```
170
    void Polynomial::UpdateCoefficients(uint64* coefs)
    {
        if (m_type == POLY_SUM) {
            m_p1->UpdateCoefficients(coefs);
175         m_p2->UpdateCoefficients(coefs);
        } else if (m_type == POLY_TERM) {
            coefs[m_xpower*(global_maxy+1)+m_ypower]+=m_coef;
            update_maxx = __max(update_maxx, m_xpower);
            update_maxy = __max(update_maxy, m_ypower);
180         update_minx = __min(update_minx, m_xpower);
            update_miny = __min(update_miny, m_ypower);
        } else if (m_type == POLY_MULTITERM) {
            for (int x=m_minx; x<=m_maxx; x++) {
                for (int y=m_miny; y<=m_maxy; y++) {
185                 if (this->GetCoefficient(x,y) != 0) {
                        // Potential overrun if invariants fail
                        if ( (x+m_xpower<0) || (x+m_xpower>global_maxx) || (y<0) || (y>
                            global_maxy) ) {
                            fprintf (stderr, "Overrun in UpdateCoefficients: %d,%d - %d
                                - %d,%d - %d,%d,%d,%d\n",
                                x, y,
190                             m_xpower,
                                global_maxx, global_maxy,
                                m_minx, m_maxx, m_miny, m_maxy);
                        }
                        // x+m_xpower to implement MultiplyByX for MULTITERM
195                     coefs[(x+m_xpower)*(global_maxy+1)+y] += this->GetCoefficient(x
                            ,y);
                        update_maxx = __max(update_maxx, x+m_xpower);
                        update_maxy = __max(update_maxy, y);
                        update_minx = __min(update_minx, x+m_xpower);
                        update_miny = __min(update_miny, y);
200                 }
                }
            }
        }
        else {
205         // This is an error!
            fprintf(stderr, "Error in UpdateCoefficient\n");
        }
    }

210 Polynomial* Polynomial::Clone()
    {
        if (m_type == POLY_TERM) {
            return new Polynomial(m_coef, m_xpower, m_ypower);
        } else if (m_type == POLY_MULTITERM) {
215         uint64* newcoefs = (uint64*)malloc((m_maxx-m_minx+1)*(m_maxy-m_miny+1)*
                sizeof(uint64));
            memcpy(newcoefs, m_coefs, (m_maxx-m_minx+1)*(m_maxy-m_miny+1)*sizeof(uint64
                ));
            return new Polynomial(newcoefs, this->m_minx, this->m_miny, this->m_maxx,
                this->m_maxy, this->m_xpower);
        } else {
            Polynomial* p1 = m_p1->Clone();
220         Polynomial* p2 = m_p2->Clone();
            return new Polynomial(m_type, p1, p2);
        }
```

```
}

void Polynomial::MultiplyByX()
{
    if (m_type == POLY_TERM) {
        m_xpower++;
    }
    else if (m_type == POLY_SUM) {
        m_p1->MultiplyByX();
        m_p2->MultiplyByX();
    }
    else if (m_type == POLY_MULTITERM) {
        m_xpower++;
    }
}

void Polynomial::SetHeight()
{
    if (m_p1->m_height>m_p2->m_height) {
        m_height = m_p1->m_height+1;
    } else {
        m_height = m_p2->m_height+1;
    }
}

uint64 Polynomial::GetCoefficient(int x, int y)
{
    if (this->m_type != POLY_MULTITERM) {
        fprintf (stderr, "GetCoefficient called on non-multi-term (%d,%d)\n", x, y)
            ;
        return -1;
    }
    if ( (x>m_maxx) || (x<m_minx) || (y>m_maxy) || (y<m_miny) ) {
        return 0;
    } else {
        return m_coefs[(x-m_minx)*(m_maxy-m_miny+1)+(y-m_miny)];
    }
}

bool Polynomial::SetCoefficient(int x, int y, uint64 coef)
{
    if ( (x>m_maxx) || (x<m_minx) || (y>m_maxy) || (y<m_miny) ) {
        return false;
    } else {
        m_coefs[(x-m_minx)*(m_maxy-m_miny+1)+(y-m_miny)] = coef;
        return true;
    }
}

uint64 Polynomial::AddCoefficient(int x, int y, uint64 coef)
{
    if ( (x>m_maxx) || (x<m_minx) || (y>m_maxy) || (y<m_miny) ) {
        return 0;
    } else {
        m_coefs[(x-m_minx)*(m_maxy-m_miny+1)+(y-m_miny)] += coef;
        return m_coefs[(x-m_minx)*(m_maxy-m_miny+1)+(y-m_miny)];
    }
}
```

D The PolyStash class

```
// $Header: /home/cvs/cvsroot/MSVC/TuttePol/PolyStash.cpp,v 1.11 2006/12/20 23:14:21 hkragh Exp $

#include "StdAfx.h"
#include "PolyStash.h"

#include <stdlib.h>


PolyStash::PolyStash()
{
    m_stash = (Polynomial**)malloc(STASHSIZE*sizeof(Polynomial*));
    // All stash is set to NULL (cannot be securely done with calloc)
    for (STASHINDEX i=0; i<STASHSIZE; i++) {
        m_stash[i] = NULL;
    }
    m_actives = 0;
}

PolyStash::~PolyStash()
{
    for (STASHINDEX i=0; i<STASHSIZE; i++) {
        if (m_stash[i] != NULL) {
            delete m_stash[i];
        }
    }
    free (m_stash);
}

bool PolyStash::CheckStash(STASHINDEX key)
{
    if ((key!=0) && (LookupPoly(key)!=NULL)) {
        return true;
    } else {
        return false;
    }
}

Polynomial* PolyStash::GetPoly(STASHINDEX key)
{
    if (key != 0) {
        return LookupPoly(key)->Clone();
    } else {
        fprintf(stderr, "Trying to lookup key 0.\n");
        return NULL;
    }
}

Polynomial* PolyStash::SetStash(STASHINDEX key, Polynomial *p)
{
    if ( p == NULL) {
        return NULL;
    }

    if (key == 0) {
        return p;
    }
```

```
    if ( m_stash[key] == NULL ) {
        // The polynomium does not exist in the stash: Add it (in simple form) and
            return.
60
        m_stash[key] = p->Simplify();

        m_actives++;
        if (m_actives % 256 == 0) {
65          float f = 100*(float)m_actives / STASHSIZE;
            fprintf(stderr, "Stash at %2.1f%% (%ld)\n", f, m_actives);
        }

        return p;
70  }
    else {
        // The polynomium already exists in the stash. This should not happen!
        fprintf(stderr, "Trying to reset stash (%d)\n", key);
        return p;
75  }
}

Polynomial* PolyStash::LookupPoly(STASHINDEX key)
{
80  return m_stash[key];
}
```

E StdAfx.h: 64-bit words

```
// stdafx.h : include file for standard system include files,
//  or project specific include files that are used frequently, but
//      are changed infrequently
//

#if !defined(AFX_STDAFX_H__F3B46BE4_5853_405D_A965_DDF970339F23__INCLUDED_)
#define AFX_STDAFX_H__F3B46BE4_5853_405D_A965_DDF970339F23__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000


// TODO: reference additional headers your program requires here
#include <stdlib.h>

#ifndef __max
#define __max(a,b)  ((a)>(b) ? (a) : (b))
#endif

#ifndef __min
#define __min(a,b)  ((a)<(b) ? (a) : (b))
#endif


#if defined(_WIN32)
#define int64 __int64
#define uint64 __int64
#elif defined(unix)
#define int64 long long int
#define uint64 unsigned long long int
#endif


//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the
    previous line.

#endif // !defined(AFX_STDAFX_H__F3B46BE4_5853_405D_A965_DDF970339F23__INCLUDED_)
```

F Makefile for Linux

```
all:    tuttepol-linux

tuttepol-linux: *.cpp *.h
    g++ *.cpp -o tuttepol-linux
```