Matice v NTL

Obsah

1 Moduly na prácu s maticami v knižnici NTL

Na prácu s maticami je možné v NTL použiť viacero modulov. Každý z nich má k dispozícii iné funkcie a je vhodný na použitie na iný účel. Základné moduly, ktoré pracujú s maticami sú *mat_GF2, mat_RR, mat_ZZ*. Existujú aj ďalšie moduly, s ktorými môžeme operovať v rámci matíc: mat_GF2E, mat_ZZ_p, mat_ZZ_pE, mat_lzz_p, mat_lzz_pE.

- Trieda *mat_GF2*: Matice nad GF(2), čiže trieda narába s celými číslami modulo 2. Táto trieda zahŕňa základné aritmetické operácie, výpočet determinantu, výpočet inverznej matice, riešenie nesingulárnych systémov (systems of linear equations), a úpravu matíc pomocou Gausovej eliminácie.
- Trieda *mat_GF2E*: Trieda narába s rozšíreným poľom/okruhom nad GF2: Táto trieda má tie isté metódy ako GF2.
- Trieda *mat_RR*: Matice nad RR, čiže trieda narába s reálnymi číslami. Táto trieda zahŕňa základné aritmetické operácie, výpočet determinantu, výpočet inverznej matice, riešenie nesingulárnych systémov (systems of linear equations).
- Trieda *mat_ZZ*: Matice nad ZZ, čiže trieda narába s veľkými celými číslami. Táto trieda má tie isté metódy ako mat_RR.
- Trieda *mat_ZZ_p, mat_ZZ_pE, mat_lzz_p, mat_lzz_pE*: Pri ZZ_p sa jedná o veľké celé čísla modulo p, pri ZZ_pE sa jedná o rozšírenie poľa/okruhu nad ZZ_p, pri lzz_p sa jedná o celé čísla mod p s jednoduchou presnosťou (single precision), pri lzz_pE sa jedná o rozšírenie poľa/okruhu nad lzz_p. Všetky tieto triedy majú rovnaké metódy ako matice nad GF2.
- Trieda *mat_poly_ZZ, mat_poly_ZZ_p, mat_poly_lzz_p*: Rutiny na výpočet charakteristických polynómov matíc sú mat_poly_ZZ, mat_poly_ZZ_p, mat_poly_lzz_p.

Všetky názorné príklady budú znázornené modulom mat_GF2. V prípade, keď sa jedná o výnimku, to bude označené pri danej metóde či funkcii.

2 Vytváranie matíc

2.1 Deklarácia matice

Deklarácia matice: Vytvorí sa matica M rozmerov 0 x 0 nad T. Pričom T môže byť ZZ,RR atď...

```
mat_T M;
```

2.2 Konštruktor matice

Inicializuje sa n x m matica M, pričom T môže byť ZZ,RR atď...

```
mat_T M(INIT_SIZE_TYPE, long n, long m);
```

2.3 Nastavenie rozmerov matice

Nastavenie matice na n riadkov krát m stĺpcov:

```
void SetDims(long n, long m);
```

Nastaví M na n x m. Ak sa zmení počet stĺpcov (m), pôvodné miesto sa uvoľní a nové miesto v pamäti sa inicializuje a matica sa realokuje. Napríklad nastavenie matice M na 10x20: *M.SetDims(10, 20);*

Na riadok môžeme pristupovať cez M[i], kde i je index žiadaného riadku s indexovaním od 0, alebo ako M(i), pri indexovaní od 1. A na prvok v matici môžeme pristupovať protredníctvom M[i][j], (indexovanie od 0), alebo M(i, j), (indexovanie od 1).

Rozmery matice je možné zmeniť. Pokiaľ sa nemení počet stĺpcov, matica sa iba zmení na velkosti ako vektor, a žiadne informácie nie su stratené. Ak sa zmení počet stĺpcov, tak je matica porušená a následne sa nová vytvorí.

2.4 kill

```
void kill();
```

M.kill() Uvolní pamäť a nastaví maticu na 0 x 0.

2.5 NumRows

```
long NumRows() const;
```

M.NumRows() Vráti počet riadkov M.

2.6 NumCols

```
long NumCols() const;
```

M.NumCols() Vráti počet stĺpcov M.

2.7 position

```
long position(const vec_T& a) const;
```

Vráti index vektoru a, ak sa v matici nachádza. Vráti -1 ak sa tam nevyskytuje; ekvivaletne rep(*this).position(a).

2.8 swap

```
void swap(mat_T& X, mat_T& Y);
```

Vymenia sa X a Y (vymenenim pointrov)

2.9 MakeMatrix

```
void MakeMatrix(mat_T& x, const vec_vec_T& a);
```

Skopíruje a do x, kontrolujúc či je a "štvorcová"

2.10 get

```
GF2 get(long i, long j) const;
```

Vráti záznam z pozície (i, j) , s indexovaním od 0. Pracuje iba s GF2.

2.11 put

```
void put(long i, long j, GF2 a);
```

Zapíše na pozíciu (i, j) hodnotu a, pri indexovaní od 0. Pracuje iba s GF2.

2.12 conv

```
void conv(mat_GF2& X, const vec_vec_GF2& A);
mat_GF2 to_mat_GF2(const vec_vec_GF2& A);
```

Skonvertuje vektor vec_GF2 na maticu. Pracuje iba s GF2.

3 Aritmetické operácie

3.1 Sčítanie matíc

Na sčítanie a odčítanie matíc slúžia funkcie *add*, *sub*, *negate*. Tieto funkcie sa dajú použiť, pre všetky dátové typy, ktoré v NTL reprezentujú matice (okrem mat_ZZp). Tak isto slúžia aj klasické operátory $+, -, +=, -=, ++, --$.

3.1.1 add

```
void add(mat_GF2& X, const mat_GF2& A, const mat_GF2& B); // X = A + B
```

Do premennej *mat_GF2 X*, priradí súčet matíc *mat_GF2 A* + *mat_GF2 B*.

3.1.2 sub

```
void sub(mat_GF2& X, const mat_GF2& A, const mat_GF2& B); // X = A - B
```

Do premennej *mat_GF2 X*, priradí rozdiel matíc *mat_GF2 A* - *mat_GF2 B*. Iba v prípade matíc nad GF2 a nad GF2E sa rozdiel matíc rovná súčtu matíc a teda X = A - B = A + B.

3.1.3 negate

```
void negate(mat_GF2& X, const mat_GF2& A); // X = -A
```

Do premennej *mat_GF2 X*, priradí negáciu matice *mat_GF2 A*.Iba v prípade matíc nad GF2 a nad GF2E sa negácia matice rovná matici samej a teda // X = -A = A.

3.2 Násobenie matíc

Na násobenie matíc sa používa operátor $*, *=$. Na násobenie a umocňovanie sa používajú aj funkcie *mul*, *sqr*, *power*, *inv*.

3.2.1 mul

```
void mul(mat_GF2& X, const mat_GF2& A, const mat_GF2& B); // X = A * B
```

Do premennej *mat_ GF2 X*, priradí výsledok násobenia matíc *mat_ GF2 A, const mat_ GF2 B.*

```
void mul(vec_GF2& x, const mat_GF2& A, const vec_GF2& b); // x = A * b
```

Do premennej *vec_ GF2 x*, priradí výsledok násobenia matice a vektora *mat_ GF2 A, const vec_ GF2 b.*

```
void mul(mat_GF2& X, const mat_GF2& A, GF2 b); //X = A * b
```

Do premennej *mat_ GF2 X*, priradí výsledok násobenia matice a čísla *mat_ GF2 A, GF2 b.*

3.2.2 sqr

```
void sqr(mat_GF2& X, const mat_GF2& A); // X = A*A
```

Do premennej *mat_ GF2 X* priradí druhú mocninu *mat_ GF2 A.*

3.2.3 power

```
void power(mat_GF2& X, const mat_GF2& A, long e);
```

Do premennej *mat_ GF2 X* priradí *long* A^e, e môže byť aj záporné, vtedy musí byť A nesingulárna.

3.2.4 inv

```
void inv(GF2& d, mat_GF2& X, const mat_GF2& A); // X = A^{-1}.
```

Do premennej *mat_ GF2 X* priradí inverznú maticu, ak A je matica n x n a determinant d = det(A) *GF2 d* je *rôzny* od nuly . $X = A^{-1}$

4 Riešenia matíc

Vo všetkých moduloch je možné používať na riešenie matíc funkciu *solve.* Môžeme na ne použiť funkcie, ktoré sú nápomocné pri výpočte riešenia matice: *determinant*, *transpose*, *ident*, *diag*, *gauss*, *image*, *kernel.*

4.1 determinant

```
void determinant(GF2& d, const mat_GF2& A); // d = det(A)
```

Do premennej *GF2 d*, priradí determinant matice A *mat_ GF2 A.*

4.2 transpose

```
void transpose(mat_GF2& X, const mat_GF2& A); // X = {A}^T
```

Do premennej *mat_ GF2 X*, sa transponuje matica A *mat_ GF2 A.* $X = A^T$

4.3 ident

```
void ident(mat_GF2& X, long n); // X = In
```

Do premennej *mat_ GF2 X* priradí jednotková matica veľkosti n x n. $X = I_n$

4.4 diag

```
void diag(mat_GF2& X, long n, GF2 d); // X = D
```

Do premennej *mat_ GF2 X* priradí matica veľkosti n x n s diagonálnym elemetnom *GF2 d*. Na diagonále bude mať prvky d.

4.5 gauss

```
long gauss(mat_GF2& M, long w);
```

Do premennej *mat_ GF2 M* sa priradí matica M upravená elmentárnymi riadkovými operáciami na stupňovitý tvar. Ak je zadaný voliteľný argument w, úprava matice sa zastaví, keď sa prvých w stĺpcov nachádza v stupňovitom tvare. Funkcia vráti hodnosť matice (alebo rank prvých w stĺpcov). Táto funkcia sa nevyskytuje v moduloch mat_ZZ a mat_RR.

4.6 image

```
void image(mat_GF2& X, const mat_GF2& A);
```

Riadky matice X sú bázou riadkového priestoru A. X je v stupňovitej forme.

4.7 kernel

```
void kernel(mat_GF2& X, const mat_GF2& A);
```

Vypočíta sa báza pre jadro, pričom X je riadkový vektor.

4.8 solve

```
void solve(GF2& d, vec_GF2& x, const mat_GF2& A, const vec_GF2& b);
```

A je n x n matica, b je dĺžkový vektor, ktorý musí mať kompatibilnú dimeziu(dĺžku) s maticou A, d = det(A), d != 0, vyrieši sa x*A = b.

4.9 solve1

```
void solve1(ZZ& d, vec_ZZ& x, const mat_ZZ& A, const vec_ZZ& b);
```

A je štvorcová matica, ak je singulárna, táto rutina nasvatí d=0 a skončí, inak vypočíta *d,x* ako *x*A = b*d* (d je kladné a minimálne). Treba si uvedomiť, že d je kladný deliteľ determinantu a v skutku nie je ekvivaletný s determinantom. Táto rutina je deterministiclá a používa Hensel lifting stratégiu. (Pre spätnú kompatibilitu existuje rutina s názvom HenselSolve1, ktorá iba zavolá solve1). Táto funkcia je dostupná iba pre modul mat_ZZ.

5 Testovanie a porovnávanie matíc

Vo všetkých moduloch je možné matice testovať. Umožnujú to buď klasické operátory == a ! =, alebo funkcie *IsIdent, IsDiag, IsZero.*

5.1 IsIdent

```
long IsIdent(const mat_GF2& A, long n);
```

Testuje, či A je n x n jednotková matica.

5.2 IsDiag

```
long IsDiag(const mat_GF2& A, long n, long d);
```

Testuje, či X je n x n diagonálna matica s diagonálnym elementom (d mod 2).

5.3 IsZero

```
long IsZero(const mat_GF2& A);
```

Funkcia testuje, či je matica A nulová.

5.4 Clear

```
void clear(mat_ZZ_pE& a);
```

Funkcia zapíše do matice nuly a nezmení jej rozmery.

6 Utility routines

V tejto časti sa nachádza niekoľko užitočných funkcií, ktoré sa vyskytujú iba v niektorých moduloch. Sú tu obsiahnuté rôzne funkcie, ktoré nie su súčasťou modulov NTL, ktoré sa vyrobili na základe požiadaviek zákazníka.

6.1 CRT

Predpokladajme, že $m_1, m_2..., m_r$ sú navzájom nesúdeliteľné prirodzené čísla, $m_i >= 2$ pre i=1 až r. Potom každá sústava rovníc:
$x \equiv a_1 (mod\ m_1)$
$x \equiv a_2 (mod\ m_2)$
.
.
.
$x \equiv a_k (mod\ m_r)$
má riešenie x a toto riešenie je jednoznačne určené v modulo $M = m_1 m_2 ... m_r$.

```
long CRT(mat_ZZ& a, ZZ& prod, const mat_zz_p& A);
```

Funkcia sa nachádza iba v module mat_ZZ. Incremental Chinese Remaindering: Ak *p* je *zz_p* modulus s (p, prod) = 1; CRT vypočíta *a'* ako *a' = a mod prod* a *a' = A mod p*, s koeficientami z intervalu *(-p*prod/2, p*prod/2)*; Nastaví *a := a', prod := p*prod*, a vráti 1 ak sa hodnota *a* zmenila.

6.2 CharPoly

```
void CharPoly(ZZX& f, const mat_ZZ& M);
```

f = characteristický polynóm matice M

7 HNF

Hnf je samostatný modul, ktorý obsahuje iba jednu rutinu na vypočítanie Hermite Normal Form. HNF je forma stupňovitej matice (hornej). Matica M je v hermitovej normálnej forme ak, je horná trojuholníková a žiaden z prvkov nie je záporný. Ak je M štvorcová, tak neobsahuje žiadne nuly na diagonále. Nech je daná štvorcová nesigulárna matica nxn A, potom existuje unimodulárna nxn U a nxn M(v HNF), že $AU = M$.

```
void HNF(mat_ZZ& W, const mat_ZZ& A, const ZZ& D);
```

Vstupná matica A má rozmery n x n hodnosti m (t.j. n >= m), a D je násobok determinantu sita L obaleného (zahŕňajúceho) riadkami A. W je vypočítaná Hermitová normálna forma matice A, teda W je jedinečná matica m x m, ktorej riadky obalujú L.

8 LLL

Rutiny sú určené pre redukciu bázy zväzu, zahŕňajúc obe metódy: exaktné varianty(pomalé ale presné), varianty s pohyblivou desatinou čiarkou(rýchle ale aproximujúce). Nech $f_1 a f_n$ sú lineárne nezávislé vektory z R^n potom zväz je suma $(u_i * f_i)$, pričom i je 1 až n, kde u_i patrí do Z. Tento zväz je plnohodnostný (full-ranked). f_i su vektory, ktoré tvoria bázu zväzu. determinant zväzu je absolútna hodnota determinantu matice tvorenej z f_i.

8.1 Exaktné aritmetické varianty

LLL redukcia

```
long LLL(ZZ& det2, mat_ZZ& B, long verbose = 0);
long LLL(ZZ& det2, mat_ZZ& B, mat_ZZ& U, long verbose = 0);

long LLL(ZZ& det2, mat_ZZ& B, long a, long b, long verbose = 0);
long LLL(ZZ& det2, mat_ZZ& B, mat_ZZ& U, long a, long b, long verbose = 0);
```

B je matica m x n (m riadkov n-dĺžkových vektorov). "m"môže byť <,=,> ako "n", riadky nemusia byť lineárne nezávislé. B je transformovaná na LLL-redukovanú bázu zväzu a návratová hodnota funkcie je hodnosť "r"matice B. Prvých m-r riadkov matice B je nulových.

Na matici B sú vykonané elementárne riadkové operácie tak, aby nenulové riadky novej B vytvorili LLL-redukovanú bázu pre zväz obalený riadkami starej B. Defaultny redukčný parameter je $\Delta = \frac{3}{4}$, čo znamená že štvorec dĺžky prvého nenulového vektora bázy, nie je väčší než 2^{r-1} krát ako najkratší vektor vo zväze.

det2 je vypočítaný ako štvorec determinantu zväzu. (odmocnia z det2 je celé číslo len, keď r = n)

V druhej verzii je U nastavené na transformačnú maticu, takže U je unimodulárna m x m matica. U * stará B = nová B. (prvých m-r riadkov z U tvorí bázu /ako pri zväze/ pre jadro starej B)

Tretia a štvrtá verzia povoľujú ľubovoľný redukčný parameter Δ = a/b, kde 1/4 < a/b <= 1, kde *a* a *b* sú celé kladné čísla. Pre redukovanú bázu parametrom delta, štvorec dĺžky prvého nenulového bázového vektora nie je väčší ako $\frac{1}{(\Delta-\frac{1}{4})^{r-1}}$ násobok najkratšieho vektora vo zväze. (Schnorr a Euchner nižšie)

Variácie

```
long LLL_plus(vec_ZZ& D, mat_ZZ& B, long verbose = 0);
long LLL_plus(vec_ZZ& D, mat_ZZ& B, mat_ZZ& U, long verbose = 0);

long LLL_plus(vec_ZZ& D, mat_ZZ& B, long a, long b, long verbose = 0);
long LLL_plus(vec_ZZ& D, mat_ZZ& B, mat_ZZ& U, long a, long b,
              long verbose = 0);
```

Toto sú rutiny, ktoré objasnia viac informácií o redukovanej báze. Pokiaľ je r hodnosť B, potom D je vektor dĺžky r + 1, teda D[0] = 1 a pre i = 1 až r, D[i]/D[i-1] je rovnaké ako štvorec dĺžky i-teho vektora Gram-Schmidtovej bázy zhodujúcej sa s (nenulovými) riadkami LLL redukovanej bázy B. Vo všeobecnosti, D[r] je ekvivaletné hodnote det2 získaného obyčajnými LLL rutinami.

8.1.1 Počítanie obrazov a jadier

```
long image(ZZ& det2, mat_ZZ& B, long verbose = 0);
long image(ZZ& det2, mat_ZZ& B, mat_ZZ& U, long verbose = 0);
```

Počítanie obrazu B použitím "cheap"verzie LLL: poskytuje zvyčajnú "redukciu veľkosti", a iba vymení vektory pokiaľ sú nájdené lineárne závislosti. Ako aj v predošlých LLL rutinách, návratová hodnota je hodnosť r matice B a prvých m-r riadkov budú nula. U je unimodulárna (unimodulárna = celočíselná matica ktorej determinant je +-0,1) m x m matica kde platí, že U * stará B = nová B. det2 determinant zväzu umocnený na druhú (v praxi je do absolútna hodnota determinantu vektorov, ktoré tvoria bázu zväzu,a ešte umecnené na druhú).

Treba s pripomenúť, že prvých m-r riadkov U tvoria bázu (ako zväz) pre jadro starej B. Toto je adekvátny praktický algoritmus na počítanie jadier. Niekto môže tiež aplikovať image() na jadro, kvôli kratším bázovým vektorom pre jadrá (nie sú lineárne závislosti, ale redukcia veľkosti môže pomôcť). Pre každé kratšie jadrové bázové vektory sa môže aplikovať LLL().

8.1.2 Hľadanie vektora vo zväze

```
long LatticeSolve(vec_ZZ& x, const mat_ZZ& A, const vec_ZZ& y, long reduce=0);
```

Toto testuje, či pre dané A a *y* existuje x, aby *x * A = y*. Ak áno x je riešením a návratová hodnota je 1, inak x ostane nezmenené a návratová hodnota je 0.

Voliteľný parameter *reduce* ovláda kvalitu výsledného vektora. Pokiaľ riadky matice A sú lineárne závislé, je mnoho riešení, pokiaľ nejaké existujú. Hodnota reduce obmedzenia ovláda množstvo námahy, ktorá vstupuje do hľadania krátkeho výsledného vektora.

reduce = 0: žiadne špecifické úsilie nevstupuje do hľadanie krátkeho riešenia
reduce = 1: Jednoduchý algoritmus redukcie veľkosti beží na jadre(A), toto je rýchle a môže postúpiť ku kratšiemu riešeniu ako defaultne, ale nie veľmi blízko k optimu.
reduce = 2: LLL algoritmus beží na jadre(A), toto môže značne pomalšie ako ostatné voľby, ale postupuje k riešeniam, ktoré sú evidentne bližšie k optimu. Presnejšie, ak jadro(A) má hodnosť k, potom štvorec dĺžky získaného riešenia, nie je väčší ako $\max(1, 2^{k-2})$ krát oproti optimálnemu riešeniu. Toto núti použiť jemnú variáciu Babai's algoritmu , algoritmu najbližšieho vektora.

Pravdaže, pokiaľ riadky A sú lineárne nezávislé, potom hodnota obmedzenia je irelevantná a teda riešenie (pokiaľ existuje) je jedinečné.

8.2 Varianty s pohyblivou desatinou čiarkou

Existuje množstvo LLL variantov s pohyblivou desatinnou čiarkou: je možné si vybrať presnosť, ortogonalizačnú stratégiu a redukčnú podmienku. Široká rozmanitosť výberu sa môže zdať mätúca, pozri návod "Ako si vybrať".
Presnosť

- FP - double
- QP - quad_float
- XD - xdouble
- RR

Ortogonalizačná stratégia

- Klasická Gramm-Schmidt ortogonalizácia používa klasické metódy pre výpočet G-S orto. Je rýchla ale náchylná na problémy so stabilitou. Táto stratégia bola prvý krát predložená Snorchlom a Eunuchom.
- Givensova ortogonalizácia je o čosi pomalšia, ale vo všeobecnosti stabilnejšia no najpreferovanejšia ortogonalizačná stratégia

Redukčná podmienka

- Klasciká LLL redukčná podmienka
- Bloková Korkin-Zolotarevova redukcia. Je pomalá, ale postupuje ku kvalitnejšej báze.

8.3 LLL rutiny - syntax

```
long
[G_]LLL_{FP,QP,XD,RR} (mat_ZZ& B, [ mat_ZZ& U, ] double delta = 0.99,
                       long deep = 0, LLLCheckFct check = 0, long verbose = 0);
```

Zátvorky [...] naznačujú voliteľnosť a ... naznačujú výber z viacerých alternatív. Návratová hodnota je hodnosť matice B. (ale if check ! = 0). Voliteľný prefix G_ indikuje, že sú používané Givensove rotácie, v opačnom prípade je použitý Gram-Schmidt. FP, QP, XD, RR určujú presnosť. Ak je zvolený parameter U, vypočíta sa transponovaná matica : U* stará_B = nová_B. Voliteľný parameter "delta"je redukčný parameter, ktorý môže byť nastavený v rozsahu od 0.5, vrátane, po 1. Nastavením na hodnotu blízku k jednej sa skracuje vektor a tiež sa zlepšuje stabilita a zvyšuje čas. Doporučená hodnota je 0.99. Voliteľný parameter "deep"je možné nastaviť na hocaké celé číslo, ktoré umožňuje "hĺbkové vkladanie"riadku k do riadku i do hĺbky i alebo do hĺbky k-i. Voliteľný parameter "check"je funkciou, ktorá sa volá po každej redukcii veľkosti aktuálnym riadkom ako argumentom. Ak táto funkcia vráti nenulovú hodnotu, LLL procedúra je okamžite ukončená. Je možné, že niektoré lineárne závislosti ostanú skryté kvôli veľkej hodnosti matice. V každom prípade, nájdené nulové riadky budú umiestnené na začiatok. Check argument (pokiaľ nie je nulový) by mal byť rutinou s konštantou vec_ZZ& ako argumentom a návratovou hodnotou z check funkcie typu long. LLLCheckFct je definovaný: typedef long (*LLLCheckFct)(const vec_ZZ&);

Voliteľný parameter "verbose"sa nastavuje kvoli výpisu výstupov z rutiny. Stavové hlásenie je vypisované zakaždým a báza je voliteľne ukladaná do súboru. Správanie je možné ovládať prostredníctvom globálnych premenných:

```
extern char *LLLDumpFile;
extern double LLLStatusInterval;
```

8.4 BKZ rutiny - syntax

```
long
[G_]BKZ_{FP,QP,QP1,XD,RR} (mat_ZZ& B, [ mat_ZZ& U, ] double delta=0.99,
                           long BlockSize=10, long prune=0,
                           LLLCheckFct check = 0, long verbose = 0);
```

Tieto funkcie sú ekvivalentné s rutinami LLL okrem toho, že sa na nich aplikuje bloková Korkin-Zolotarevová redukcia. Rozdielne parametre:

Voliteľný parameter "BlockSize"špecifikuje veľkosť blokov v redukcii. Vysoké hodnoty dávajú krátke vektory, ale čas výpočtu narastá exponeneciálne s dĺžkou bloku. BlockSize by mala byť v rozmedzí od 2 po počet riadkov B. Voliteľný parameter "prune"je nastavovaný na kladné čísla na vyvolanie Volume Heuristic. Toto značne redukuje výpočtový čas a teda dovolí väčšie bloky, ale kvalita redukcie nie je taká dobrá. Vyššie hodnoty "prune"znamenajú lepšiu kvalitu na úkor času. Ak je prune nastavené na 0, tak sa pruningovanie (obmedzovanie) nevykonáva. Pre bloky veľkosti väčšej alebo rovnej ako 30 sa obyčajne používa prune v rozsahu 9 až 16. QP1 variant používa quad_float presnosť na výpočet Gramm-Schimdta, ale používa dvojitú presnosť v prehľadávacej fáze algoritmu redukcie blokov. Toto vyzerá byť vhodné pre mnohé účely a rýchlejšie ako QP, ktoré používa quad_float presnosť.

8.5 Ako si vybrať?

Zrejme nik nevie ako v skutočnosti funguje LLL algoritmus. Teoretické analýzy sú dlhou cestou opisu, čo sa v skutočnosti deje v praxi. Výber najlepšej varianty pre konkrétnu aplikáciu sa koná spôsobom pokusov a omylov. Prvá vec ktorú treba skúsiť je LLL_FP. Je to najrýchlejšia rutina a adaptabilná na mnohé aplikácie. Ak sú problémy s presnosťou, s najväčšou pravdepodobnosťou vyskočí ErrorMessagBox (warning-relaxing redcution). Pokiaľ nastali nejaké problémy s pretečením, treba zmeniť príliš veľké číslo (o čom sme informovaný). Pokiaľ nenastala ani jedna možnosť, ďalšia vec ktorú môžeme skúsiť je G_LLL_FP ktoré používa síce pomalšie, ale stabilnejšie a spoľahlivejšie Givensove rotácie. Toto priblíženie je príjemná vlastnosť, že čísla ostávajú menšie, takže je menej pravdepodobné pretečenie. Pokiaľ sú stále problémy s presnosťou s G_LLL_FP, treba skúsiť LLL_QP alebo G_LLL_QP, ktoré používa kvadratickú presnosť a ďalej LLL_XD or G_LLL_XD.

Všetky spomínané moduly sa vzťahujú na BKZ varianty. Pri veľmi veľkých maticiach treba skúsiť najprv pomocou G_LLL_FP alebo LLL_XD na redukciu veľkosti a potom použiť BKZ varianty.