|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **НУЛП, ІКНІ, САП** | | Тема | оцінка | підпис |
| СПКм-12 | 1 | **ECJ** |  |  |
| Цедуляк Т. Б. | |
| № залікової: | |
| Методи нечіткої логіки та еволюційні алгоритми при автоматизованому проектуванні | | Викладач: | |
| Кривий Р.З. | |

1. **Мета роботи**

Ознайомитись з можливостями ECJ. Здобути навики використання програмного продукту для вирішення поставлених задач.

1. **Короткі теоретичні відомості**

**Про ECJ**

ECJ це фрейморк спрямований на роботу з еволюційними алгоритмами, який написаний на Java. Фреймворк був розроблений для потреб великих, експерементальних обчислень і надає інструменти, які забезпечують безліч популярних алгоритмів, але з особливим акцентом у бік генетичного програмування. ECJ є фреймворком з відкритим вихідним кодом.

ECJ уже налічує більше десяти років ефективної та стабільної роботи і є стабільною платформою, в якій було виявлено відносно мало серйозних помилок протягом багатьох років. Його конструкція легко розширюється новими юнітами, в тому числі і юнітами для роботи з багатокритеріальними алгоритми оптимізації, острівних моделей, стаціонарні і стратегії еволюції методи, та багато інших. Система широко використовується в генетичному програмуванні і є досить популярним в цілому світі.

ECJ був розроблений для великих проектів і надає широкий спектр послуг. Основним джерелом натхнення ECJ є Lil-gp, якій він багато чим зобов'язаний. Робота над ECJ почалася восени 1998 року після того, як з досвідом Lil-gp в еволюційно модельованих команд робо футболу. Цей проект потребував значних модифікацій (Lil-gp), зокрема для виконання паралельних оцінок, багаторазове використання модулів, і суворої типізації. Такі аргументи ясно дали зрозуміти, що Lil-pg не може розвиватись далі без значних зусилью. ECJ призначений для щонайменше, десяти ічного терміну корисного використання, і він добре зарекомендував себе і досі.

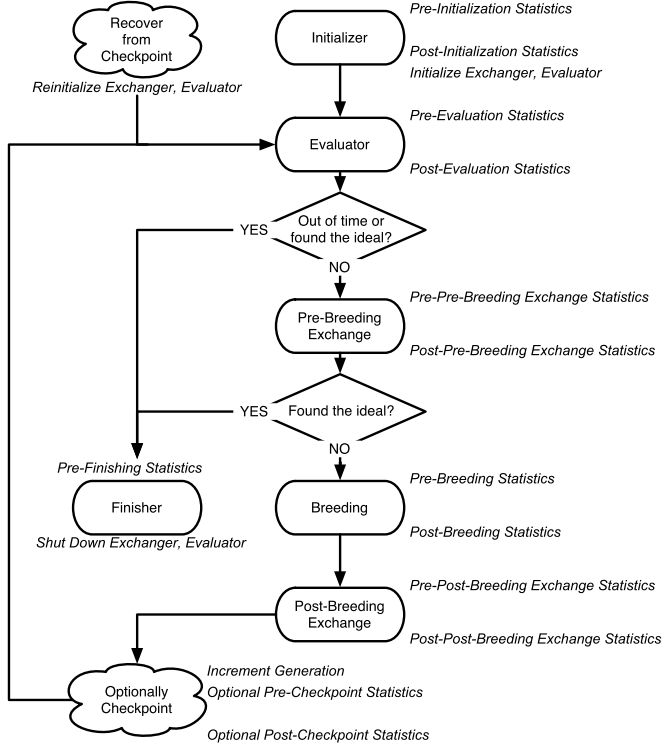


Рис. 1. Високорівневі операції класу SimpleEvolutionState ECJ.

**Огляд**

**Петлі верхнього рівня**

ECJ наслідує весі стани еволюційного алгоритму з одного примірника підкласу EvolutionState. Це дозволяє ECJ серіалізовати увест стан системи до контрольної точки, точки відновлення. На малюнку 1.1 показана петля верхнього рівня простого нащадка EvolutionState у ітерації циклу між розведенням і оцінкою, після кожного з них. Статистична петля викликається перед і після кожного періоду розмноження, оцінки і обміну, а також до і після ініціалізації популяції і "очищення" (очищення до виходу з програми).

Конкретні варіанти цих Singleton об'єктів визначаються експериментатором, хоча ми пропонуємо варіанти, які виконують загальні їх набори.

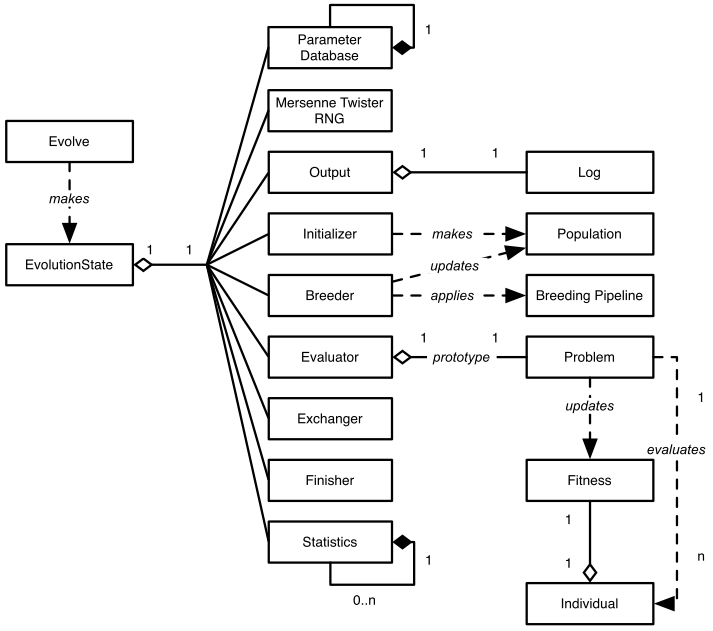


Рис. 1.2. Оператори верхнього рівня і підкласи в EvolutionState, а також їх зв'язок з окремими об'єктами.

**Оцінка.** Оцінювач проводить оцінку популяції шляхом пропускання одного або кількох індивідів до проблеми підкласу, який оцінювач клонував від свого прототипу.

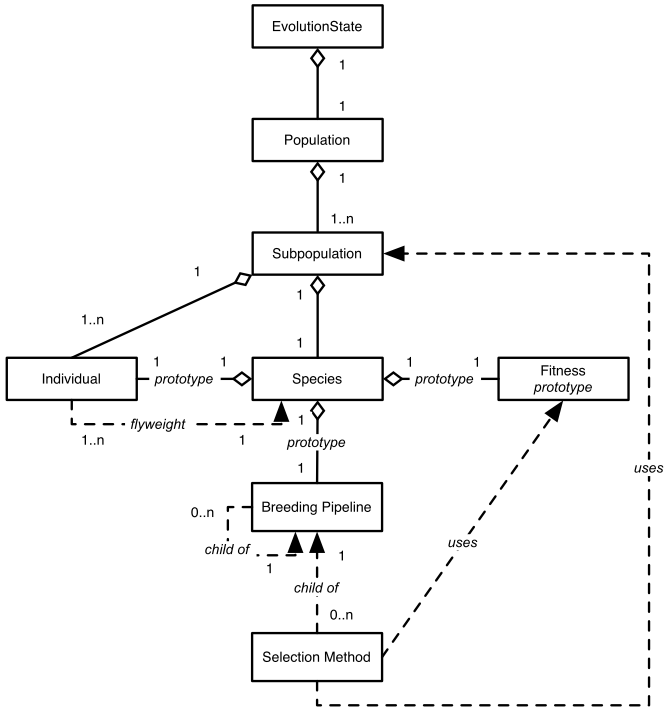


Рис. 1.3. Об'єкти верхнього рівня даних що використовується в процесі еволюції.

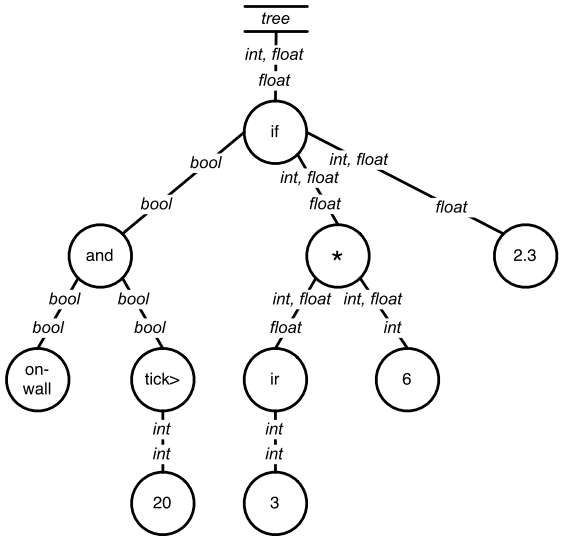


Рис. 1.4. Типізоване генетичне дерево синтаксичного аналізу.

Оцінка також може бути зроблено в багатопотоковому режимі, без блокування, використовуючи окремий потік для кожної проблеми. Індивіди можуть також піддаватися багаторазовій оцінці.

У більшості проектів з використанням ECJ, основне завдання полягає в тому, щоб побудувати відповідний підклас Problem. Завдання Problem полягає в тому, щоб оцінити індивідів і встановити їх придатність до певних вимог. Класи проблема також можуть повідомити, коли було виявлено ідеального індивіда.

Якість генератора випадкових чисел має важливе значення для системи стохастичної оптимізації.

**З коробки ECJ забезпечує підтримку наступних алгоритмів:**

• Generational алгоритми: (μ, λ) і (μ + λ) Еволюція стратегій, генетичний алгоритм.

• Просторово-похідний еволюційний алгоритм

• Випадкові зхрещування

• Багатоцільова оптимізація, в тому числі NSGA-II і алгоритмів SPEA2.

• Оцінка багатозадачності і розведення.

• Паралельний синхронний і асинхронний алгоритм роботи на мережі комп'ютерів.

• Внутрішні синхронні Island Моделі всередині популяції.

• Асинхронна еволюція.

• Гнучка еволюція, під час якої віддалені машини опрацьовують свої власні міні-еволюційні процеси.

• Внутрішні синхронні Island моделі всередині популяції.

• Метаеволюції

• Велика кількість селекцй і операторів

1. **Процес встановлення ACJ**

Фреймворк ACJ написано на Java, це означає, що він не вимагає будь-якого додаткового програмного забезпечення. Все що необхідно, це мати інстальованим Java JDK 1.5 або більш пізню версію. Вихідний код ACJ поставляється в пакеті ecj.22.jar, який можна завантажити з наступного поситання <https://cs.gmu.edu/~eclab/projects/ecj/>.

Нижче буде представлено спосіб інсталяції ACJ використовуючи Eclipse IDE.

Після завантаження фреймворк матиме виляд JAR бібліотеки рис. 1.5.

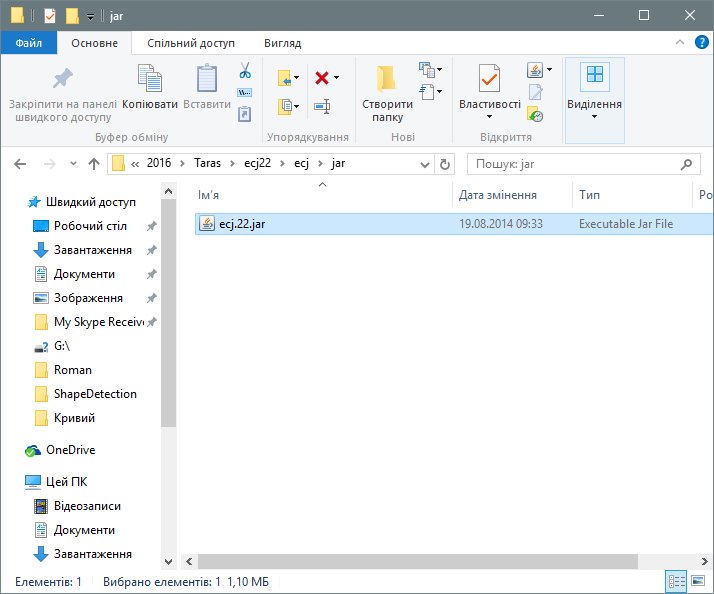


Рис. 1.5. JAR бібліотека ACJ після завантаження

Наступним кроком є підключення ACJ до середовища Eclipse IDE. Для цього необхідно виконати наступні кроки:

* Виберіть *File* → *Create* → *New Java Project*.
* Введіть ім'я проекту (наприклад, ACJproject) і натисніть на кнопку *Finish*.
* Натисніть правою кнопкою миші на папці з проектом (в провіднику Eclipse) та оберіть *Properties* (рис. 1.6).
* Оберіть розділ *Java Build Pat*.
* В меню праворуч натисніть кнопку *Add External JARs..* (рис. 1.7)
* Вкажіть шлях до папки, що містить JAR бібліотеку ACJ та натисніть відкрити (рис. 1.8).

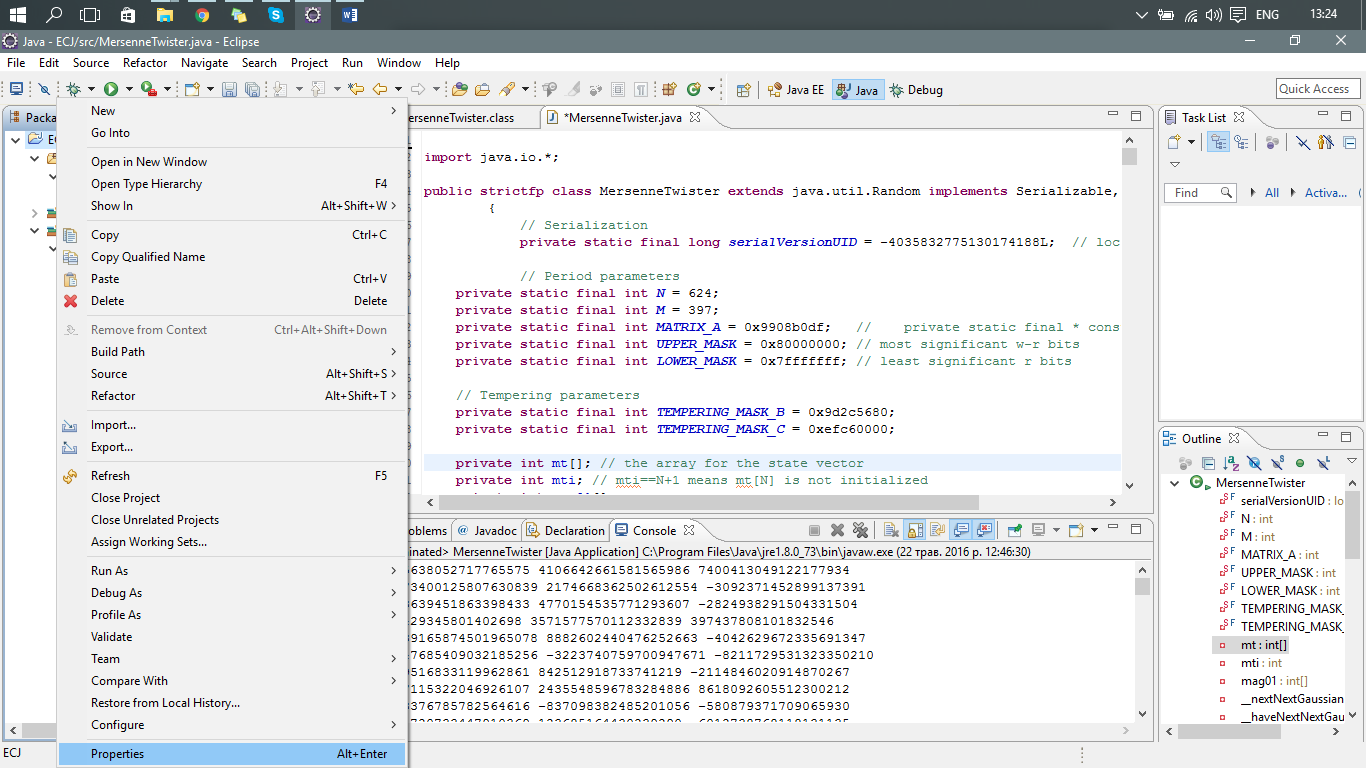


Рис. 1.6. Вікно *Project Properties*

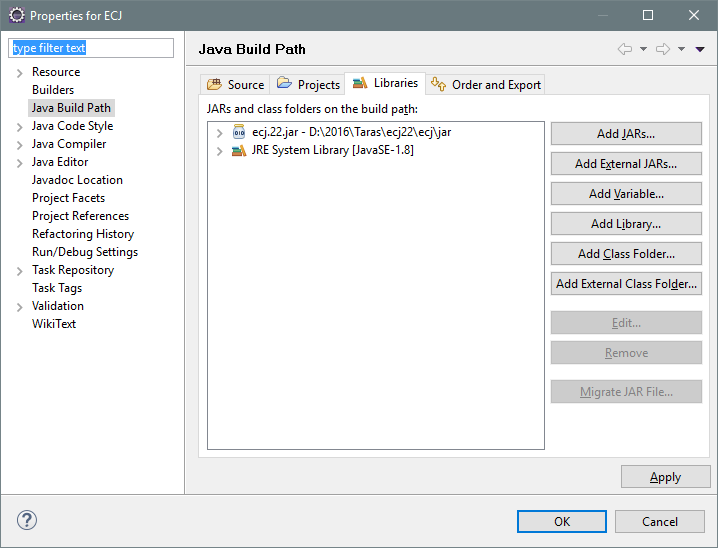


Рис. 1.7. Вікно *Java Build Pat*

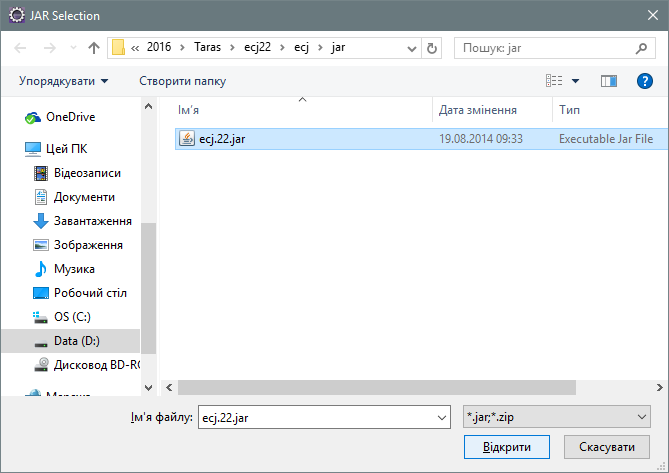


Рис. 1.8. *JAR Selection*

Мною було обране тестове завдання MersenneTwister в якості прикладу, код якого представлений в додатку А. А результати його виконання представленні в додатку Б. Нижче, на рис. 1.9 зображено вікно Eclipse після підключення фреймворку ACJ та виконання програми MersenneTwister.

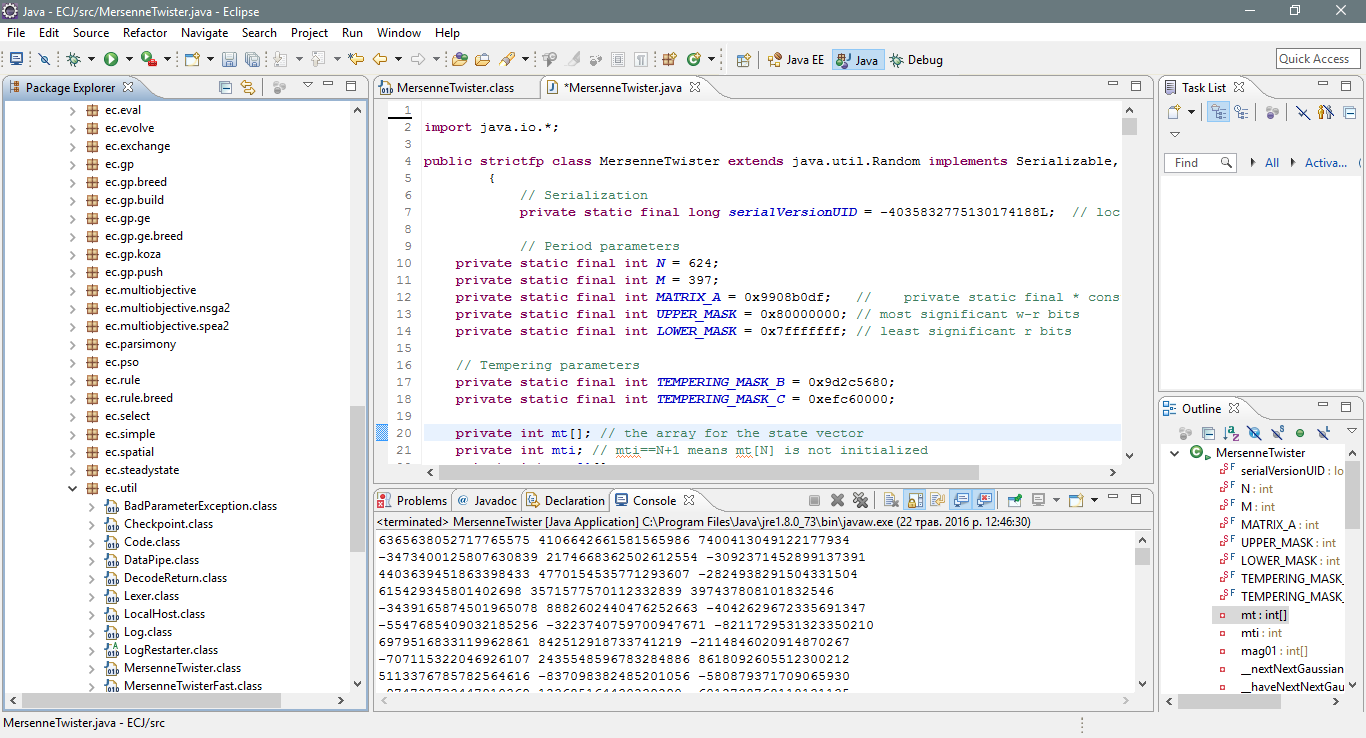


Рис. Представлення реалізації програми MersenneTwister в ACJ

**Висновок:**

**Додаток А.**

**import** java.io.\*;

**public** **strictfp** **class** MersenneTwister **extends** java.util.Random **implements** Serializable, Cloneable

{

// Serialization

**private** **static** **final** **long** ***serialVersionUID*** = -4035832775130174188L; // locked as of Version 15

// Period parameters

**private** **static** **final** **int** ***N*** = 624;

**private** **static** **final** **int** ***M*** = 397;

**private** **static** **final** **int** ***MATRIX\_A*** = 0x9908b0df; // private static final \* constant vector a

**private** **static** **final** **int** ***UPPER\_MASK*** = 0x80000000; // most significant w-r bits

**private** **static** **final** **int** ***LOWER\_MASK*** = 0x7fffffff; // least significant r bits

// Tempering parameters

**private** **static** **final** **int** ***TEMPERING\_MASK\_B*** = 0x9d2c5680;

**private** **static** **final** **int** ***TEMPERING\_MASK\_C*** = 0xefc60000;

**private** **int** mt[]; // the array for the state vector

**private** **int** mti; // mti==N+1 means mt[N] is not initialized

**private** **int** mag01[];

// a good initial seed (of int size, though stored in a long)

//private static final long GOOD\_SEED = 4357;

/\* implemented here because there's a bug in Random's implementation

of the Gaussian code (divide by zero, and log(0), ugh!), yet its

gaussian variables are private so we can't access them here. :-( \*/

**private** **double** \_\_nextNextGaussian;

**private** **boolean** \_\_haveNextNextGaussian;

/\* We're overriding all internal data, to my knowledge, so this should be okay \*/

**public** Object clone()

{

**try**

{

MersenneTwister f = (MersenneTwister)(**super**.clone());

f.mt = (**int**[])(mt.clone());

f.mag01 = (**int**[])(mag01.clone());

**return** f;

}

**catch** (CloneNotSupportedException e) { **throw** **new** InternalError(); } // should never happen

}

**public** **boolean** stateEquals(Object o)

{

**if** (o==**this**) **return** **true**;

**if** (o == **null** || !(o **instanceof** MersenneTwister))

**return** **false**;

MersenneTwister other = (MersenneTwister) o;

**if** (mti != other.mti) **return** **false**;

**for**(**int** x=0;x<mag01.length;x++)

**if** (mag01[x] != other.mag01[x]) **return** **false**;

**for**(**int** x=0;x<mt.length;x++)

**if** (mt[x] != other.mt[x]) **return** **false**;

**return** **true**;

}

/\*\* Reads the entire state of the learning\_mersenne\_rand.MersenneTwister RNG from the stream \*/

**public** **void** readState(DataInputStream stream) **throws** IOException

{

**int** len = mt.length;

**for**(**int** x=0;x<len;x++) mt[x] = stream.readInt();

len = mag01.length;

**for**(**int** x=0;x<len;x++) mag01[x] = stream.readInt();

mti = stream.readInt();

\_\_nextNextGaussian = stream.readDouble();

\_\_haveNextNextGaussian = stream.readBoolean();

}

/\*\* Writes the entire state of the learning\_mersenne\_rand.MersenneTwister RNG to the stream \*/

**public** **void** writeState(DataOutputStream stream) **throws** IOException

{

**int** len = mt.length;

**for**(**int** x=0;x<len;x++) stream.writeInt(mt[x]);

len = mag01.length;

**for**(**int** x=0;x<len;x++) stream.writeInt(mag01[x]);

stream.writeInt(mti);

stream.writeDouble(\_\_nextNextGaussian);

stream.writeBoolean(\_\_haveNextNextGaussian);

}

/\*\*

\* Constructor using the default seed.

\*/

**public** MersenneTwister()

{

**this**(System.*currentTimeMillis*());

}

/\*\*

\* Constructor using a given seed. Though you pass this seed in

\* as a long, it's best to make sure it's actually an integer.

\*/

**public** MersenneTwister(**long** seed)

{

**super**(seed); /\* just in case \*/

setSeed(seed);

}

/\*\*

\* Constructor using an array of integers as seed.

\* Your array must have a non-zero length. Only the first 624 integers

\* in the array are used; if the array is shorter than this then

\* integers are repeatedly used in a wrap-around fashion.

\*/

**public** MersenneTwister(**int**[] array)

{

**super**(System.*currentTimeMillis*()); /\* pick something at random just in case \*/

setSeed(array);

}

/\*\*

\* Initalize the pseudo random number generator. Don't

\* pass in a long that's bigger than an int (Mersenne Twister

\* only uses the first 32 bits for its seed).

\*/

**synchronized** **public** **void** setSeed(**long** seed)

{

// it's always good style to call super

**super**.setSeed(seed);

// Due to a bug in java.util.Random clear up to 1.2, we're

// doing our own Gaussian variable.

\_\_haveNextNextGaussian = **false**;

mt = **new** **int**[***N***];

mag01 = **new** **int**[2];

mag01[0] = 0x0;

mag01[1] = ***MATRIX\_A***;

mt[0]= (**int**)(seed & 0xffffffff);

mt[0] = (**int**) seed;

**for** (mti=1; mti<***N***; mti++)

{

mt[mti] =

(1812433253 \* (mt[mti-1] ^ (mt[mti-1] >>> 30)) + mti);

/\* See Knuth TAOCP Vol2. 3rd Ed. P.106 for multiplier. \*/

/\* In the previous versions, MSBs of the seed affect \*/

/\* only MSBs of the array mt[]. \*/

/\* 2002/01/09 modified by Makoto Matsumoto \*/

// mt[mti] &= 0xffffffff;

/\* for >32 bit machines \*/

}

}

/\*\*

\* Sets the seed of the learning\_mersenne\_rand.MersenneTwister using an array of integers.

\* Your array must have a non-zero length. Only the first 624 integers

\* in the array are used; if the array is shorter than this then

\* integers are repeatedly used in a wrap-around fashion.

\*/

**synchronized** **public** **void** setSeed(**int**[] array)

{

**if** (array.length == 0)

**throw** **new** IllegalArgumentException("Array length must be greater than zero");

**int** i, j, k;

setSeed(19650218);

i=1; j=0;

k = (***N***>array.length ? ***N*** : array.length);

**for** (; k!=0; k--)

{

mt[i] = (mt[i] ^ ((mt[i-1] ^ (mt[i-1] >>> 30)) \* 1664525)) + array[j] + j; /\* non linear \*/

// mt[i] &= 0xffffffff; /\* for WORDSIZE > 32 machines \*/

i++;

j++;

**if** (i>=***N***) { mt[0] = mt[***N***-1]; i=1; }

**if** (j>=array.length) j=0;

}

**for** (k=***N***-1; k!=0; k--)

{

mt[i] = (mt[i] ^ ((mt[i-1] ^ (mt[i-1] >>> 30)) \* 1566083941)) - i; /\* non linear \*/

// mt[i] &= 0xffffffff; /\* for WORDSIZE > 32 machines \*/

i++;

**if** (i>=***N***)

{

mt[0] = mt[***N***-1]; i=1;

}

}

mt[0] = 0x80000000; /\* MSB is 1; assuring non-zero initial array \*/

}

/\*\*

\* Returns an integer with <i>bits</i> bits filled with a random number.

\*/

**synchronized** **protected** **int** next(**int** bits)

{

**int** y;

**if** (mti >= ***N***) // generate N words at one time

{

**int** kk;

**final** **int**[] mt = **this**.mt; // locals are slightly faster

**final** **int**[] mag01 = **this**.mag01; // locals are slightly faster

**for** (kk = 0; kk < ***N*** - ***M***; kk++)

{

y = (mt[kk] & ***UPPER\_MASK***) | (mt[kk+1] & ***LOWER\_MASK***);

mt[kk] = mt[kk+***M***] ^ (y >>> 1) ^ mag01[y & 0x1];

}

**for** (; kk < ***N***-1; kk++)

{

y = (mt[kk] & ***UPPER\_MASK***) | (mt[kk+1] & ***LOWER\_MASK***);

mt[kk] = mt[kk+(***M***-***N***)] ^ (y >>> 1) ^ mag01[y & 0x1];

}

y = (mt[***N***-1] & ***UPPER\_MASK***) | (mt[0] & ***LOWER\_MASK***);

mt[***N***-1] = mt[***M***-1] ^ (y >>> 1) ^ mag01[y & 0x1];

mti = 0;

}

y = mt[mti++];

y ^= y >>> 11; // TEMPERING\_SHIFT\_U(y)

y ^= (y << 7) & ***TEMPERING\_MASK\_B***; // TEMPERING\_SHIFT\_S(y)

y ^= (y << 15) & ***TEMPERING\_MASK\_C***; // TEMPERING\_SHIFT\_T(y)

y ^= (y >>> 18); // TEMPERING\_SHIFT\_L(y)

**return** y >>> (32 - bits); // hope that's right!

}

/\* If you've got a truly old version of Java, you can omit these

two next methods. \*/

**private** **synchronized** **void** writeObject(ObjectOutputStream out)

**throws** IOException

{

// just so we're synchronized.

out.defaultWriteObject();

}

**private** **synchronized** **void** readObject (ObjectInputStream in)

**throws** IOException, ClassNotFoundException

{

// just so we're synchronized.

in.defaultReadObject();

}

/\*\* This method is missing from jdk 1.0.x and below. JDK 1.1

includes this for us, but what the heck.\*/

**public** **boolean** nextBoolean() {**return** next(1) != 0;}

/\*\* This generates a coin flip with a probability <tt>probability</tt>

of returning true, else returning false. <tt>probability</tt> must

be between 0.0 and 1.0, inclusive. Not as precise a random real

event as nextBoolean(double), but twice as fast. To explicitly

use this, remember you may need to cast to float first. \*/

**public** **boolean** nextBoolean (**float** probability)

{

**if** (probability < 0.0f || probability > 1.0f)

**throw** **new** IllegalArgumentException ("probability must be between 0.0 and 1.0 inclusive.");

**if** (probability==0.0f) **return** **false**; // fix half-open issues

**else** **if** (probability==1.0f) **return** **true**; // fix half-open issues

**return** nextFloat() < probability;

}

/\*\* This generates a coin flip with a probability <tt>probability</tt>

of returning true, else returning false. <tt>probability</tt> must

be between 0.0 and 1.0, inclusive. \*/

**public** **boolean** nextBoolean (**double** probability)

{

**if** (probability < 0.0 || probability > 1.0)

**throw** **new** IllegalArgumentException ("probability must be between 0.0 and 1.0 inclusive.");

**if** (probability==0.0) **return** **false**; // fix half-open issues

**else** **if** (probability==1.0) **return** **true**; // fix half-open issues

**return** nextDouble() < probability;

}

/\*\* This method is missing from JDK 1.1 and below. JDK 1.2

includes this for us, but what the heck. \*/

**public** **int** nextInt(**int** n)

{

**if** (n<=0)

**throw** **new** IllegalArgumentException("n must be positive, got: " + n);

**if** ((n & -n) == n)

**return** (**int**)((n \* (**long**)next(31)) >> 31);

**int** bits, val;

**do**

{

bits = next(31);

val = bits % n;

}

**while**(bits - val + (n-1) < 0);

**return** val;

}

/\*\* This method is for completness' sake.

Returns a long drawn uniformly from 0 to n-1. Suffice it to say,

n must be > 0, or an IllegalArgumentException is raised. \*/

**public** **long** nextLong(**long** n)

{

**if** (n<=0)

**throw** **new** IllegalArgumentException("n must be positive, got: " + n);

**long** bits, val;

**do**

{

bits = (nextLong() >>> 1);

val = bits % n;

}

**while**(bits - val + (n-1) < 0);

**return** val;

}

/\*\* A bug fix for versions of JDK 1.1 and below. JDK 1.2 fixes

this for us, but what the heck. \*/

**public** **double** nextDouble()

{

**return** (((**long**)next(26) << 27) + next(27))

/ (**double**)(1L << 53);

}

/\*\* Returns a double in the range from 0.0 to 1.0, possibly inclusive of 0.0 and 1.0 themselves. Thus:

<p><table border=0>

<th><td>Expression<td>Interval

<tr><td>nextDouble(false, false)<td>(0.0, 1.0)

<tr><td>nextDouble(true, false)<td>[0.0, 1.0)

<tr><td>nextDouble(false, true)<td>(0.0, 1.0]

<tr><td>nextDouble(true, true)<td>[0.0, 1.0]

</table>

<p>This version preserves all possible random values in the double range.

\*/

**public** **double** nextDouble(**boolean** includeZero, **boolean** includeOne)

{

**double** d = 0.0;

**do**

{

d = nextDouble(); // grab a value, initially from half-open [0.0, 1.0)

**if** (includeOne && nextBoolean()) d += 1.0; // if includeOne, with 1/2 probability, push to [1.0, 2.0)

}

**while** ( (d > 1.0) || // everything above 1.0 is always invalid

(!includeZero && d == 0.0)); // if we're not including zero, 0.0 is invalid

**return** d;

}

/\*\* A bug fix for versions of JDK 1.1 and below. JDK 1.2 fixes

this for us, but what the heck. \*/

**public** **float** nextFloat()

{

**return** next(24) / ((**float**)(1 << 24));

}

/\*\* Returns a float in the range from 0.0f to 1.0f, possibly inclusive of 0.0f and 1.0f themselves. Thus:

<p><table border=0>

<th><td>Expression<td>Interval

<tr><td>nextFloat(false, false)<td>(0.0f, 1.0f)

<tr><td>nextFloat(true, false)<td>[0.0f, 1.0f)

<tr><td>nextFloat(false, true)<td>(0.0f, 1.0f]

<tr><td>nextFloat(true, true)<td>[0.0f, 1.0f]

</table>

<p>This version preserves all possible random values in the float range.

\*/

**public** **float** nextFloat(**boolean** includeZero, **boolean** includeOne)

{

**float** d = 0.0f;

**do**

{

d = nextFloat(); // grab a value, initially from half-open [0.0f, 1.0f)

**if** (includeOne && nextBoolean()) d += 1.0f; // if includeOne, with 1/2 probability, push to [1.0f, 2.0f)

}

**while** ( (d > 1.0f) || // everything above 1.0f is always invalid

(!includeZero && d == 0.0f)); // if we're not including zero, 0.0f is invalid

**return** d;

}

/\*\* A bug fix for all versions of the JDK. The JDK appears to

use all four bytes in an integer as independent byte values!

Totally wrong. I've submitted a bug report. \*/

**public** **void** nextBytes(**byte**[] bytes)

{

**for** (**int** x=0;x<bytes.length;x++) bytes[x] = (**byte**)next(8);

}

/\*\* For completeness' sake, though it's not in java.util.Random. \*/

**public** **char** nextChar()

{

// chars are 16-bit UniCode values

**return** (**char**)(next(16));

}

/\*\* For completeness' sake, though it's not in java.util.Random. \*/

**public** **short** nextShort()

{

**return** (**short**)(next(16));

}

/\*\* For completeness' sake, though it's not in java.util.Random. \*/

**public** **byte** nextByte()

{

**return** (**byte**)(next(8));

}

/\*\* A bug fix for all JDK code including 1.2. nextGaussian can theoretically

ask for the log of 0 and divide it by 0! See Java bug

<a href="http://developer.java.sun.com/developer/bugParade/bugs/4254501.html">

http://developer.java.sun.com/developer/bugParade/bugs/4254501.html</a>

\*/

**synchronized** **public** **double** nextGaussian()

{

**if** (\_\_haveNextNextGaussian)

{

\_\_haveNextNextGaussian = **false**;

**return** \_\_nextNextGaussian;

}

**else**

{

**double** v1, v2, s;

**do**

{

v1 = 2 \* nextDouble() - 1; // between -1.0 and 1.0

v2 = 2 \* nextDouble() - 1; // between -1.0 and 1.0

s = v1 \* v1 + v2 \* v2;

} **while** (s >= 1 || s==0 );

**double** multiplier = StrictMath.*sqrt*(-2 \* StrictMath.*log*(s)/s);

\_\_nextNextGaussian = v2 \* multiplier;

\_\_haveNextNextGaussian = **true**;

**return** v1 \* multiplier;

}

}

/\*\*

\* Tests the code.

\*/

**public** **static** **void** main(String args[])

{

**int** j;

MersenneTwister r;

// CORRECTNESS TEST

// COMPARE WITH http://www.math.keio.ac.jp/matumoto/CODES/MT2002/mt19937ar.out

r = **new** MersenneTwister(**new** **int**[]{0x123, 0x234, 0x345, 0x456});

System.***out***.println("Output of learning\_mersenne\_rand.MersenneTwister with new (2002/1/26) seeding mechanism");

**for** (j=0;j<1000;j++)

{

// first, convert the int from signed to "unsigned"

**long** l = (**long**)r.nextInt();

**if** (l < 0 ) l += 4294967296L; // max int value

String s = String.*valueOf*(l);

**while**(s.length() < 10) s = " " + s; // buffer

System.***out***.print(s + " ");

**if** (j%5==4) System.***out***.println();

}

// SPEED TEST

**final** **long** SEED = 4357;

**int** xx; **long** ms;

System.***out***.println("\nTime to test grabbing 100000000 ints");

r = **new** MersenneTwister(SEED);

ms = System.*currentTimeMillis*();

xx=0;

**for** (j = 0; j < 100000000; j++)

xx += r.nextInt();

System.***out***.println("Mersenne Twister: " + (System.*currentTimeMillis*()-ms) + " Ignore this: " + xx);

System.***out***.println("To compare this with java.util.Random, run this same test on MersenneTwisterFast.");

System.***out***.println("The comparison with Random is removed from learning\_mersenne\_rand.MersenneTwister because it is a proper");

System.***out***.println("subclass of Random and this unfairly makes some of Random's methods un-inlinable,");

System.***out***.println("so it would make Random look worse than it is.");

// TEST TO COMPARE TYPE CONVERSION BETWEEN

// MersenneTwisterFast.java AND learning\_mersenne\_rand.MersenneTwister.java

System.***out***.println("\nGrab the first 1000 booleans");

r = **new** MersenneTwister(SEED);

**for** (j = 0; j < 1000; j++)

{

System.***out***.print(r.nextBoolean() + " ");

**if** (j%8==7) System.***out***.println();

}

**if** (!(j%8==7)) System.***out***.println();

System.***out***.println("\nGrab 1000 booleans of increasing probability using nextBoolean(double)");

r = **new** MersenneTwister(SEED);

**for** (j = 0; j < 1000; j++)

{

System.***out***.print(r.nextBoolean((**double**)(j/999.0)) + " ");

**if** (j%8==7) System.***out***.println();

}

**if** (!(j%8==7)) System.***out***.println();

System.***out***.println("\nGrab 1000 booleans of increasing probability using nextBoolean(float)");

r = **new** MersenneTwister(SEED);

**for** (j = 0; j < 1000; j++)

{

System.***out***.print(r.nextBoolean((**float**)(j/999.0f)) + " ");

**if** (j%8==7) System.***out***.println();

}

**if** (!(j%8==7)) System.***out***.println();

**byte**[] bytes = **new** **byte**[1000];

System.***out***.println("\nGrab the first 1000 bytes using nextBytes");

r = **new** MersenneTwister(SEED);

r.nextBytes(bytes);

**for** (j = 0; j < 1000; j++)

{

System.***out***.print(bytes[j] + " ");

**if** (j%16==15) System.***out***.println();

}

**if** (!(j%16==15)) System.***out***.println();

**byte** b;

System.***out***.println("\nGrab the first 1000 bytes -- must be same as nextBytes");

r = **new** MersenneTwister(SEED);

**for** (j = 0; j < 1000; j++)

{

System.***out***.print((b = r.nextByte()) + " ");

**if** (b!=bytes[j]) System.***out***.print("BAD ");

**if** (j%16==15) System.***out***.println();

}

**if** (!(j%16==15)) System.***out***.println();

System.***out***.println("\nGrab the first 1000 shorts");

r = **new** MersenneTwister(SEED);

**for** (j = 0; j < 1000; j++)

{

System.***out***.print(r.nextShort() + " ");

**if** (j%8==7) System.***out***.println();

}

**if** (!(j%8==7)) System.***out***.println();

System.***out***.println("\nGrab the first 1000 ints");

r = **new** MersenneTwister(SEED);

**for** (j = 0; j < 1000; j++)

{

System.***out***.print(r.nextInt() + " ");

**if** (j%4==3) System.***out***.println();

}

**if** (!(j%4==3)) System.***out***.println();

System.***out***.println("\nGrab the first 1000 ints of different sizes");

r = **new** MersenneTwister(SEED);

**int** max = 1;

**for** (j = 0; j < 1000; j++)

{

System.***out***.print(r.nextInt(max) + " ");

max \*= 2;

**if** (max <= 0) max = 1;

**if** (j%4==3) System.***out***.println();

}

**if** (!(j%4==3)) System.***out***.println();

System.***out***.println("\nGrab the first 1000 longs");

r = **new** MersenneTwister(SEED);

**for** (j = 0; j < 1000; j++)

{

System.***out***.print(r.nextLong() + " ");

**if** (j%3==2) System.***out***.println();

}

**if** (!(j%3==2)) System.***out***.println();

System.***out***.println("\nGrab the first 1000 longs of different sizes");

r = **new** MersenneTwister(SEED);

**long** max2 = 1;

**for** (j = 0; j < 1000; j++)

{

System.***out***.print(r.nextLong(max2) + " ");

max2 \*= 2;

**if** (max2 <= 0) max2 = 1;

**if** (j%4==3) System.***out***.println();

}

**if** (!(j%4==3)) System.***out***.println();

System.***out***.println("\nGrab the first 1000 floats");

r = **new** MersenneTwister(SEED);

**for** (j = 0; j < 1000; j++)

{

System.***out***.print(r.nextFloat() + " ");

**if** (j%4==3) System.***out***.println();

}

**if** (!(j%4==3)) System.***out***.println();

System.***out***.println("\nGrab the first 1000 doubles");

r = **new** MersenneTwister(SEED);

**for** (j = 0; j < 1000; j++)

{

System.***out***.print(r.nextDouble() + " ");

**if** (j%3==2) System.***out***.println();

}

**if** (!(j%3==2)) System.***out***.println();

System.***out***.println("\nGrab the first 1000 gaussian doubles");

r = **new** MersenneTwister(SEED);

**for** (j = 0; j < 1000; j++)

{

System.***out***.print(r.nextGaussian() + " ");

**if** (j%3==2) System.***out***.println();

}

**if** (!(j%3==2)) System.***out***.println();

}

}

**Додаток Б.**

6365638052717765575 4106642661581565986 7400413049122177934

-3473400125807630839 2174668362502612554 -3092371452899137391

4403639451863398433 4770154535771293607 -2824938291504331504

615429345801402698 3571577570112332839 397437808101832546

-3439165874501965078 8882602440476252663 -4042629672335691347

-5547685409032185256 -3223740759700947671 -8211729531323350210

6979516833119962861 842512918733741219 -2114846020914870267

-707115322046926107 2435548596783284886 8618092605512300212

5113376785782564616 -837098382485201056 -580879371709065930

-974720733447910369 123685164430239290 -6012738768118131135

387552258653318972 2002317554841340205 -2025348451362310369

-1717267019372065344 -4437652990902928966 -8615210504364927422

995859545206108798 2709962729049736766 1241520865745189205

-1238714264439515509 -8033501440557479344 -6532222682645593522

881876238581421351 -5277961345826405488 -7281192570905614930

-3776964400107250213 -2687652928193355118 1312374863005774988

-2020535093058122559 1297001408891116828 -4143933745892341056

4326735238335681312 -1172558852437447245 8479591749198046926

-7846834292507729533 -4291547925022947240 -7210405337702014022

-6571457766114276893 -3646717488355631273 7411345705687457779

-4686472594104577208 7300931331231742833 -8621733087466201703

-1518662709621881734 8225687255816276416 -6190747612139684990

-6161797746987420761 -3040877477580376710 8440103324527845972

6748071886070180939 -1530366433439284350 2236011780047861059

-5581265176968275694 -6821682986853075014 4393866057515712191

-5343428293751779427 9085490805179486098 2250812349134664137

6550273300932467717 -1769051187335171960 -3373266505325715483

4910405835870739385 -4568959427839127561 9028225678053002186

6055994858434998732 -3092029655300256291 5947433603322254726

1759389798535273987 -2571333116831430891 -279584404984066384

-3160851178924957584 9027982713000769369 -735439608081577069

-2568005019275019695 6131525359614849291 8342465792883066529

6375331993483933421 5859096386474744489 9039743009023368126

-571874187176609621 4182201505122390724 5288720079188123098

1831424835238802015 5688549183587687877 7215421545997774328

3704402107261628478 -2160812387041545710 6535372533771085572

-5705606390587778064 6698091817317980349 3704482778474845128

-4715841025052782029 4527614067996796232 -2084678034712305529

-8641254999441382900 5802306571083898608 367161376762076207

7928754111839409828 -228380705644903697 9124442841410779588

-1995134152724720500 7887109670825575003 8929570470352009036

1656655753431161785 439686858753538999 -8647266643803116619

-2260049451767054030 9173611853134976406 -2100627999770601747

2009175004014327220 -2903978297542976198 -1168505951299373584

-6646571878258625440 1354904254996226415 776831558177545874

-1097152077108928519 -4118046133277191191 127748660469286495

-1488206270103475382 3506738481335271876 3034901110528981304

-316399667959282867 346312609432398402 6595958201241232678

-4816450763668362858 5332153753183096885 -349481659102749969

-8893576325091158756 -588033265692645011 8823270292940231730

6998657821113658529 -8285894697832202380 -8299226488426224536

4449912012859260630 4026201626233823415 -4629795639976936302

8787178600883982924 -2232459623122366405 -8430494315001044468

-3895853792668081719 -4824000489288686866 5559823631126180723

-7903244239717614317 -6561705314188494396 -522124275078514325

5384307763367086087 5771664364689038133 -7961774228100933760

2664335337156684532 -7233006993355396507 6323645193350230372

-2984746178200335109 558465381819300760 2311119313273658043

-2321310958676756768 -1951049954005719156 -2178957659483397662

4371633508181863141 5418390587665900070 -1207549214596898703

6450489941949983762 -6600304629970094286 6178307733894100908

5790157207755210940 1818711041833588508 6064102776817882865

-2182286170038489611 -2688737039557717996 -8724269737992155489

4939749113492443331 1057946905500651896 -4216608903150987038

9194472791282493508 -8087948199117306063 -7013132712456998831

-5566578430030884664 -8227281659632362492 2817390566183094866

3446327515090287195 -2748113997590196393 -5979295587369378064

-6315314719382783237 3804965402034598395 705707131583490881

-5941365602136129521 7310771477280615570 -6763307176090024404

-134450609290040932 1252409854003196586 -976838068745741707

8611269339724276761 -4637863394506329173 3890812039094040049

-854534634482555936 -1184897657042079874 -6220976729701259086

-6412838585658309046 5630480922867332868 5158510273336540139

498125928924024714 -5312658114792404465 2313942563564423941