МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ  
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ЛЬВІВСЬКА ПОЛІТЕХНІКА»

Кафедра САПР



**Звіт**

**до розрахункової роботи**

з курсу:

**«**Методи нечіткої логіки та еволюційні алгоритми при автоматизованому проектуванні**»**  
на тему:  
**«Дослідження системи** **EpochX»**

Виконав:  
студент гр. КНС-13

Дробко В.М.

Перевірив:  
Кривий Р.З.

Львів 2017

# **1.Вступ**

EpochX є фреймворком Java для вивчення еволюції комп'ютерних програм, отриманих з використанням генетичних алгоритмів програмування. Версія 1.1+ має 3 повністю підтримуваних представлення: сильно типізовані дерева GP, контекстно-вільну граматики GP і граматичну Evolution. Однак, розширювана структура EpochX уможливлює реалізувати абсолютно нові уявлення в рамках еволюційної структури. Широкий діапазон реалізацій передбачені для виконання вибору, ініціалізації, кросовера, мутації ..., а також багато яких з загальних тестових завдань, які використовують GP.  
 EpochX призначений в першу чергу для дослідників, що працюють по генетичній теорії програмування. Вигляд людини, яка могла би отримати вигоду з використання EpochX це ті, хто зацікавлений в розподілі глибини / довжини / розмаїття і т.д., або в іншому широкому діапазоні інших статистичних даних в перспективі. Або тих, хто хоче мати дійсно динамічну систему, в якій статистичні дані доступні в режимі реального часу і параметри можуть бути оновлені під час роботи. Оскільки EpochX є основаною на Java, необхідною умовою є основною здатністю програмувати на Java.  
 EpochX побудований розумно, щоб уникнути проблем з продуктивністю, а також має підтримку кешування. Але, він не намагається бути найшвидшою системою GP. Будучи розширюваним, гнучким він забезпечує легкий доступ до даних, що має тут набагато більше значення.

# **2. Опис EpochX**

## **Встановлення**

Завантажуваний вміст фреймворка складається з таких файлів:

     epochx-1.x.x.jar – готовий для використання фреймворк

     LICENSE - копія ліцензіq GPL і LGPL

     patch-note-1.x.x.txt - список оновлень з попереднього випуску

     Javadoc - документація JavaDoc для цієї версії

     [Завантаження src також включати первинник Java-коду і необхідні бібліотеки]

EpochX не вимагає яких-небудь спеціальних налаштувань. Використання фреймворку EpochX вимагає лишень додавання файлу до шляху класів для власного додатка Java. Це дасть змогу отримати програмний доступ до цілого ряду потужних функцій, що може бути визначено в статичному файлі властивостей.

Нижче наведено зображення додавання фреймворку EpochX до Java-проекту в системі Eclipse.

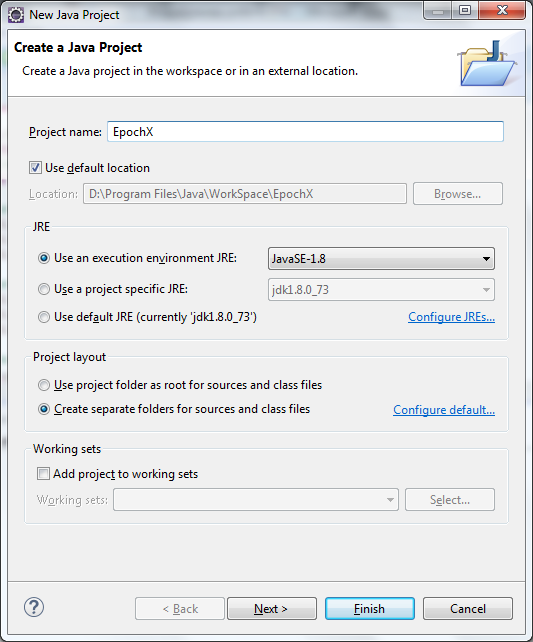


Рис. 1 Створення проекту в Eclipse під EpochX

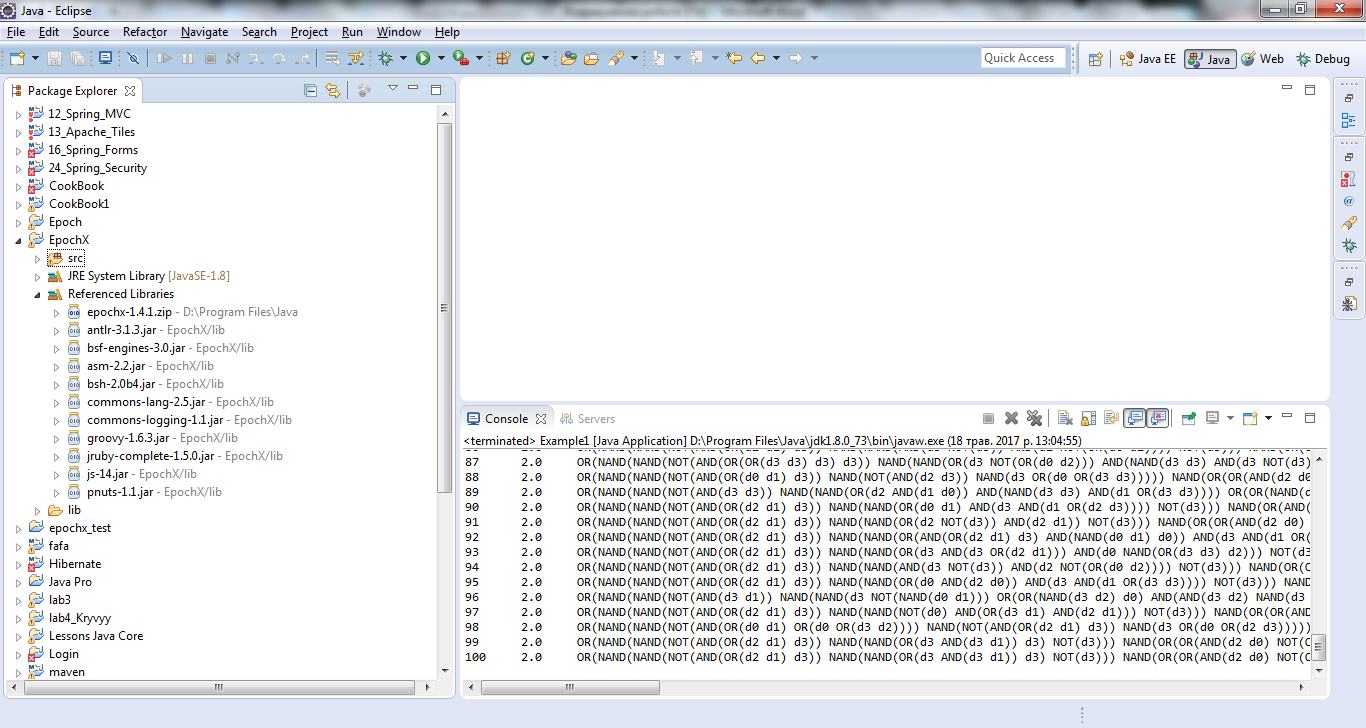
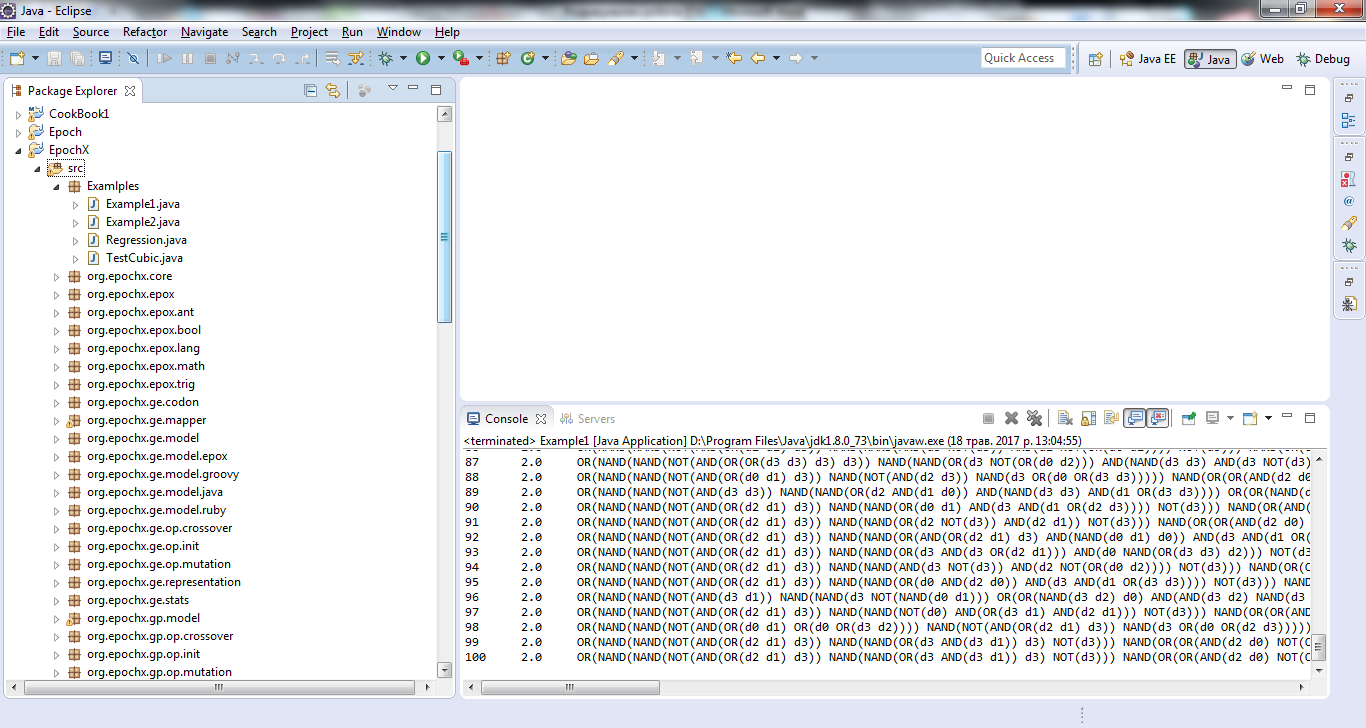


Рис. 2 Занесення jar-файлу в папку з проектом

Необхідно відкрити налаштування проекту задля підключення фреймворку, як бібліотеки.

Для зручного і коректного функціонування необхідно також підключити файли (тека org), вони також поміщаються в теку з проектом.

Оскільки, даний фреймворк є доволі старим, він має дуже багато залежностей зі іншими бібліотеками, які вже не постачаються в стандартний пакет Java, тому необхідно також додати деякі інші бібліотеки і підключати їх аналогічно. На рис. 6 видно кінцеве дерево під’єднаних файлів.



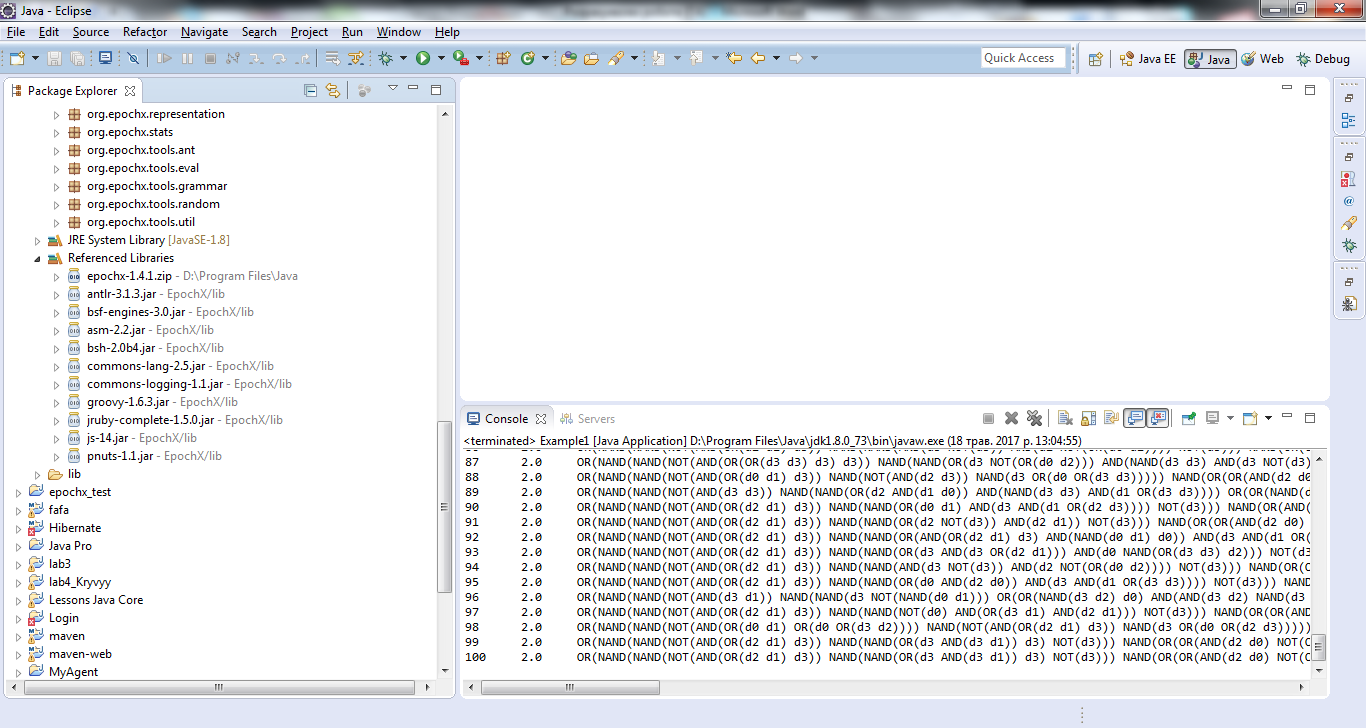


Рис. 2 Під’єднані бібліотеки

## **Огляд фреймворку**

У цьому розділі дається огляд структури API фреймворка. Більш детально можна дізнатись з JavaDoc. Структура EpochX не є складною, і легко узагальнена на діаграмі нижче (рис. 3). Всі класи в рамках вписуватися в один з чотирьох модулів, які можна було б назвати XGP, XGR, XGE і Common. XGP, модулі XGR і X містять всі класи, що відносяться до одного з уявлень - STGP, CFG-GP і GE відповідно. Будь-які класи, спільні для двох або більше уявлень будуть знайдені в рамках загального модуля.

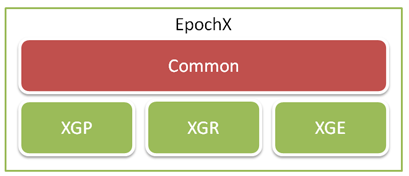


Рис. 3 Структура фреймворка

Загальний модуль містить багато найбільш важливих функціональних можливостей EpochX. Він забезпечує загальну структуру, що кожен з модулів представлення , яких є вершиною. Центральна опора каркаса є «ядро» пакету, який містить сам еволюційний алгоритм. Діаграма класів нижче (рис. 4) показує, як класи цього пакета є його частиною разом з виконанням, починаючи з моделі. Для отримання більш докладної інформації про специфіку того, що відбувається під час виконання алгоритму і порядку подій, дивіться розділ алгоритму.

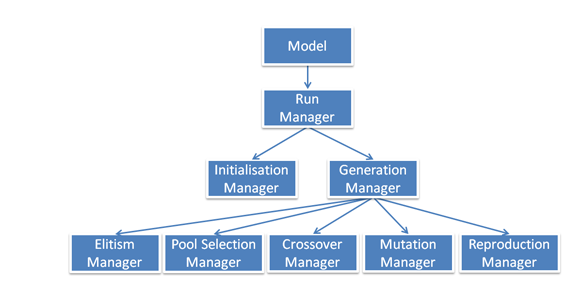


Рис. 4 Діаграма класів ядра пакету

InitialisationManager, PoolSelectionManager, CrossoverManager і MutationManager мають невелике значення для інших класів в пакеті ядра. У той час як інші класи забезпечують їх заявлену функціональність, ці з них мають порожнечі, в яку оператор повинен бути підключений для того, щоб працювати. Зверніть увагу, що ці класи не визначають кросовер, мутація і т.д. оператор. Вони просто гніздо, в якому один з цих операторів можуть бути підключені. Вони «управляти» операцією, але вони не виконують його. Набір інтерфейсів оператора визначені в org.epochx.op пакеті загального модуля. Кожен з них може бути реалізовано для виконання операції в будь-якій кількості шляхів. Оператору необхідно використовувати для установки в якості параметра на моделі. Звичайно EpochX поставляється з великою кількістю стандартних реалізацій для кожного оператора, хоча, за винятком вибору, це уявлення конкретно.

Обмеження потоків є таким, що типи операторів фіксовані, і це не представляється можливим визначити абсолютно нові типи оператора без зміни джерела фреймворка. Це обмеження буде видалено в майбутніх версіях.

Деякі з інших функцій, в Єдиному модулі є кодом життєвого циклу обробника, обробка статистики, синтаксичної граматики, перекладачі та генератори випадкових чисел. Вони всі будуть розглянуті більш детально в інших розділах.

### **XGP, XGR і XGE**

Кожен з модулів уявлень містять всі класи, необхідні для підтримки представлення програми. Для XGP це включає в себе всі різні функції і термінальні типи вузлів, які можуть бути об'єднані, щоб створити програму дерев GP. XGE і XGR з іншого боку, обидва використовують дерева розбору для представлення програм і так необхідні класи для підтримки цього можна знайти в Єдиному модулі. Кожна програма рішення кандидата виражаються в одному з цих форматів, які зберігаються всередині до примірника одного GPCandidateProgram, GRCandidateProgram або GECandidateProgram. Типи CandidateProgram зберігають деякі мета-дані про свою програму, а також надання деяких допоміжних методів для роботи з програмою.

XGP, XGR і XGE кожен з яких містить реалізації цих операторів, уявлення конкретних - кросовера, мутації і ініціалізації. З найбільш поширених операторів є, такі як SubtreeMutation для XGP і RampedHalfAndHalfInitialiser. Кожен з них реалізувати відповідний інтерфейс, наприклад, GPInitialiser, GRCrossover або GEMutation.

Інша частина структури, яка є уявленняя специфічного набору вбудованих еталонних моделей, які визначають багато із загальних проблем, яке тестується на генетичному програмуванні. Більш детальна інформації про включених моделях і як моделі в загальній роботі даються в наступному розділі моделі.

**Залежності**  
Конструкція полягає в тому, що кожен з модулів уявлень незалежний один від одного. Це означає, що можна повністю видалити XGR і XGE модулі і до сих пір повністю робочий уявлення STGP в XGP, і те ж саме для кожного з двох інших. Загальний модуль, звичайно, потрібний для всіх уявлень.

**Введення в модель**

Ключ до використання EpochX полягає в використанні моделей. Моделі програмний еквівалент для файлів параметри багатьох систем GP. Модель визначає налаштування параметрів і функцію придатності для оцінки кандидата, рішення, а також надання коштів для виконання себе.  
параметри управління

Абстрактний клас моделі (рис. 5) надає можливість обробляти всі загальні еволюційні параметри, такі як розмір популяції, або число поколінь, які необхідно виконати. Представницькі конкретні абстрактні класи успадковують від цього класу, щоб підтримати ті параметри, які є унікальними для цього подання. Наприклад, клас GEModel включає в себе параметри для обмеження кількості кодонів в хромосомі, що зробило б мало сенсу для будь-якого з класів GPModel або GRModel.  
  
 Всі компоненти перезавантажувати параметри моделі в 3-х точках під час виконання - на початку виконання, перед початком кожного запуску, і перед ним починається кожне покоління. Це припущення про структуру, яка змінює настройки модель, не відбуватиметься частіше, ніж це. Кожен компонент і оператор несе відповідальність за оновлення своїх власних налаштувань при необхідності. Тому при розробці нового модуля оператора це був вибір для розробників, щоб вирішити, чи слід реалізувати цю функцію. Всі вбудовані компоненти та оператори підтримують цю модифікацію.

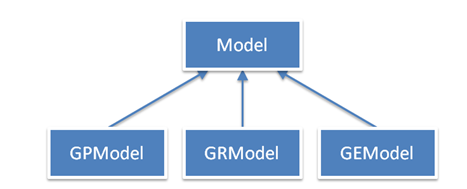


Рис. 5 Класи моделей

Кожен з класів моделей визначає ряд методів громадських мутаторів для установки значення параметра і відповідний аксессор методу для вилучення поточного встановленого значення параметра. Для повного перерахування цих методів дивіться відповідний JavaDoc.

**Фітнес-функції**

У будь-якому еволюційному алгоритмі, функція придатності побічно визначає проблему, яку необхідно вирішити, надаючи кількісну міру того, наскільки хорошим рішенням є до вирішення цієї проблеми. Кожна модель EpochX повинна забезпечувати функції придатності шляхом реалізації методи getFitness (CandidateProgram). Метод повинен обчислити і повернути фітнес бал для даного CandidateProgram. еволюційний алгоритм EpochX спирається на стандартизовані оцінки придатності, тому дуже важливо, що метод повертає менше значення для гарного рішення, ніж за поганий. Негативні значення придатності цілком прийнятні, і буде вважатися більш підходить, ніж будь-якої позитивної придатності.

### **Виконання моделі**

А також забезпечуючи всі параметри управління, модель також є точкою входу для еволюційної перспективи. Кожна модель успадковує метод, званий «прогоном», який після призову на прикладі моделі, передає нитка контролю над рамками EpochX для виконання еволюційних пробігів просили. При необхідності, контроль може бути переданий назад в додаток в певних точках пробігу (див подробиці життєвого циклу для інформації про це).

### **Еталонні моделі**

Ряд моделей приходять вбудовані в EpochX, які дозволяють легко виконання загальних тестових завдань. Ці моделі не визначають «хороші» параметри, так як він відкритий для обговорення, які параметри були б правильні параметри для установки. Замість того, щоб ці моделі просто визначити основну проблему, надаючи проблеми конкретної функції придатності, і залишаючи вибір найкращих параметрів настройки для користувача.

Наступні моделі призначені для XGP і для обох XGR і XGE в кожній з мов Java, Ruby, Groovy і Epox.

    Santa Fe – мурашиний слід

    Los Altos Hills – мурашиний слід

    Джон М'юїр – мурашиний слід

    Кубічна регресія (х + х2 + х3)  
    Quartic Regression (х + х2 + х3 + х4)  
    Sextic Regressio (х6 - 2x4 + х2)  
    Мультиплексор (скільки завгодно входів)  
    Majority (скільки завгодно входів)  
    Even Parity (скільки завгодно входів)

## **Еволюційний алгоритм**

Є кілька способів, еволюційних алгоритмів, які можуть бути організовані. Важко сказати, що один спосіб кращий або правильніший, ніж інший, але тут наводиться точний еволюційний алгоритм, що використовує EpochX. Блок-схема нижче (рис. 10) дає гарне уявлення про послідовність подій в алгоритмі. Кожен еволюційний біг з EpochX використовує цей алгоритм, незалежно від того, якої програми уявлення використовуються.

Перспектива починається з ініціалізацією нового населення з усіма програмами оцінюваного групи населення згодом. Виконання потім переходить в петлю поколінь. Кожен починає покоління з елітарністю виконується, де кращі програми поміщаються в наступне населення. Інша частина населення заповнюється в результаті генетичних операторів. Гніздовий пул з популяції входить в цикл, який триває до наступного, поеи населення не заповниться. У цьому циклі генетичний оператор випадковим чином вибирається і виконується, в результаті генетичної операції, що вставляється в наступну популяцію. Після того, як наступна популяція повна, все його програми проходять оцінку придатності. Якщо програма досягла призначеної термінації пристосованості, то прогін закінчується, в іншому випадку починається наступне покоління.

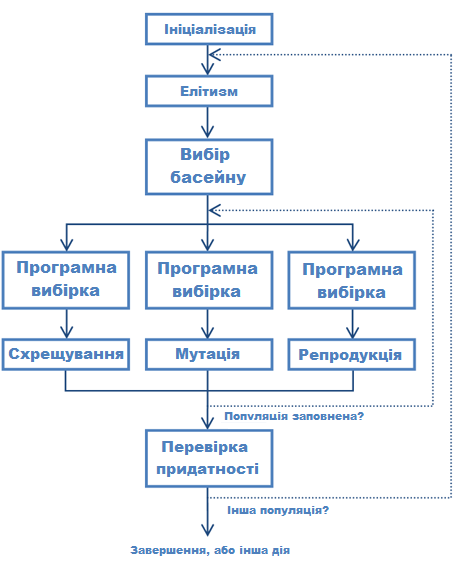


Рис. 6 Генетичний алгоритм

### **Пояснення алгоритму**

У цьому розділі буде детально, описано потік алгоритму, що досягається і за допомогою яких компонентів в коді EpochX. Весь алгоритм інкапсульовані в класі RunManager, який виконує у виконання алгоритму шляхом використання компонентів, зазначених нижче.

     Ініціалізація - управляються InitialisationManager, який витягує ініціалізатор з моделі і запитує початкову популяцію від нього. Це ініціалізатор повинен повертати список програм, як правило, це знайти б, скільки ініціалізацію і повернення з методу () в моделі getPopSize.

     Елітизм - виконується ElitismManager, який передається населення повернувся з ініціалізації. Число еліт виходять з getNoElites в моделі () метод, і список, що містить, що багато програм повертаються на стадії елітарності. Програми вибираються бути тими, з найменшими значеннями придатності. Якщо існує кілька програм рівній кращій пристосованості, то вибір між ними буде довільним. Число еліт запитуваних може дорівнювати нулю, що ефективно відключає елітарність.

     Вибір басейну - обробляються PoolSelectionManager, з реальним завданням вибору пулу, яке виконує PoolSelector витягнутого з моделі. Розмір розмноження басейну, який буде обраний, як правило, отримують з методу моделі getPoolSize (). Вибір басейну можна відключити за допомогою розміру пулу -1, і в цьому випадку саме населення використовується в якості селекційного басейну.

     Генетичні оператори - вони повторюються до наступного населення не повна, випадкового вибору, який оператор з ймовірністю відповідно до значень, що повертаються getCrossoverProbability в моделі () і getMutationProbability () методи. Розмноження відбувається залишилися ймовірності, щоб вони суму до 1,0.

         Кросовер - CrossoverManager бере на себе відповідальність за вибір дві програми з використанням ProgramSelector, отриманий з методу моделі getProgramSelector (). Потім він застосовує реалізацію кросовера, який повертає на getCrossover в моделі () метод для програм. Результат кросовера є списком з будь-якого числа програм, які вставляються в наступну популяцію.

         Мутація - The MutationManager бере на себе відповідальність за вибір однієї програми за допомогою ProgramSelector, отриманої з методу моделі getProgramSelector (), який вона проходить до реалізації Мутації повернутої по методі моделі getMutation (). В результаті мутації це програма, яка вставляється в наступну популяцію.

         Репродукування - ReproductionManager використовує ProgramSelector в моделі, щоб вибрати одну програму, яка вставляється безпосередньо в наступній популяції без маніпуляцій.

     Фітнес-оцінка - в даний час не цілком певний компонент, який управляє оцінкою придатності, обробляється всередині в RunManager. Якщо будь-яка програма досягає пристосованасті даного рівня або нижче, ніж припинення придатності зазначеного getTerminationFitness в моделі () методу, то запуск вважається успішним і закінчується. В іншому випадку виконання триватиме до більш пізнього покоління, коли припинення придатності досягається або числа поколінь, зазначених в getNoGenerations в моделі () методу досягається.

## **Події в EpochX**

EpochX має механізм обробки подій, який забезпечує спосіб прослуховування для подій життєвого циклу, таких як початок операції кросовера або в кінці покоління. Це потужний інструмент для взаємодії з середовищем в той час як вона працює. Два найбільш поширених видів використання слухачів життєвого циклу для генерації вихідного через регулярні проміжки часу, і для зміни параметрів прогону динамічно.

Всі життєві події обробляються екземпляром класу Life.Infact, ми можемо звернутися до нього як екземпляр класу життя, так як він реалізує одноплодной шаблон і так є тільки коли-небудь один об'єкт життя. Ви можете отримати посилання на цей екземпляр з будь-якого місця за допомогою виклику статичного методу Life.get (). Клас Життя має ряд методів addXxxListener для додавання слухачів для різних класів подій. Нижче наведено список стандартних методів addXxxListener.

    addCrossoverListener (CrossoverListener)  
    addElitismListener (ElitismListener)  
    addGenerationListener (GenerationListener)  
    addInitialisationListener (InitialisationListener)  
    addMutationListener (MutationListener)  
    addPoolSelectionListener (PoolSelectionListener)  
    addReproductionListener (ReproductionListener)  
    addRunListener (RunListener)

Кожен з цих методів має екземпляр слухача. Як це часто буває з шаблоном слухача, було б типовим використовувати анонімний класу, який реалізує інтерфейс слухача. Як такий, XxxAdapter абстрактний клас існує для кожного XxxListener, який реалізує цей інтерфейс, що забезпечує порожні методи для кожного з подій. Це робить його набагато більш зручним розширити XxxAdapter, перекриваючи тільки ті методи, для подій, що цікавлять вас. Наприклад, наступний код може бути використаний для прослуховування кінцевих

### **Кросовер подій**

Life.get (). AddCrossoverListener (новий CrossoverAdapter () {  
        громадського недійсний onCrossoverEnd () {  
            // ... код, який буде виконуватися в кінці кожного кросовера.  
        }  
    });

**Конфігураційні події**

На додаток до перерахованих вище класів разі, є також один інший тип події; конфиг подія. Слухачі для події конфігурації можуть бути додані таким же чином, за допомогою методу addConfigListener (ConfigListener). ConfigListener визначає тільки один метод - onConfigure. Ця подія викликала вогонь в таких випадках:

* На початку серії дослідів.
* На початку кожного прогону.
* На початку ініціалізації кожного покоління.
* При ручному звільненні.

При використанні вбудованих компонентів і операторів, забезпечених EpochX, там, ймовірно, немає необхідності використовувати конфігураційні події взагалі. Їх використання відбуватиметься у фоновому режимі. Але, якщо ви опиняєтеся реалізації нового оператора, ви можете розглянути їх використання. Детальний опис того, як йти про це буде розказано в більш пізньому підручнику, але це варто описувати поняття тут. Всі вбудовані компоненти та оператори, які вимагають параметрів мають дві форми конструкції. Вони або можуть бути побудовані з параметром самих значень, або з посиланням на модель. В останньому випадку вони отримують значення параметрів, в яких вони потребують від моделі та оновити значення параметрів, які вони використовують при кожному випадку конфігурації. Як уже згадувалося в розділі Моделі, це дозволяє моделі динамічно змінювати свої значення і отримати нове значення, яке використовується з наступного події конфігурації.  
  
 Хоча, як правило, немає необхідності, можна змусити оператор оновити свої значення параметрів з моделі частіше, ніж в кожному поколінні, запускаючи подія конфігурації вручну. Це можна зробити за допомогою наступного твердження: Life.get () fireConfigureEvent () ;.

### **Сильні і слабкі слухачі**

Насправді, методи addXxxListener, що описані вище тільки половина історії, тому що кожний з них, який приймає тільки в XxxListener, а також є ще одна версія, яка також приймає логічне значення. Логічне значення є показником, чи повинна сильна посилання буде підтримуватися до слухача, як вона зберігається в об'єкті Life. Якщо значення помилково, то тільки слабка посилання буде збережена. Більшість користувачів повинні просто використовувати стандартний метод addXxxListener, який буде використовувати сильну посилання за замовчуванням. У деяких крайніх випадках може виявитися можливим ввести витік пам'яті при додаванні слухачів і не виймаючи їх колись закінчив, використовуючи слабку посилання, вони будуть автоматично видалені, коли це можливо. Це те, що просто ігнорується більшістю користувачів, вона згадується тут в основному для повноти картини.

# **3.Приклад застосування системи**

EpochX поставляється з набором вбудованих моделей, які визначають відповідні налаштування і фітнес-функцію для загальних тестових завдань. Для початку ми будемо використовувати метод 4-парності з використанням дерева GP. Ви можете знайти навіть моделі паритету в пакеті org.epochx.gp.models.  
Створено новий клас Java, з основним методом. Усередині основного методу додали наступні рядки коду для того, щоб почати розвивати деякі програми GP.

1. GPModel model = new EvenParity(4);

2. Life.get().addGenerationListener(new GenerationAdapter(){

3. public void onGenerationEnd() {

4. Stats.get().print(StatField.GEN\_NUMBER,

5. StatField.GEN\_FITNESS\_MIN,

6. StatField.GEN\_FITTEST\_PROGRAM);

7. }

8. });

9. model.run();

                         
 Рядок 1 будує модель, рядки 2-8 визначає, які статистичні дані ми хотіли б надрукувати кожне покоління і рядок 7 виконує нашу модель. Ми могли викинути рядки 2-8, і наша модель буде як і раніше розвиватись, але не було б ніякого висновку. Якщо ви намагаєтеся запустити цей код, що Ви повинні переконатися, що ви імпортувати правильний клас EvenParity оскільки модель призначена для кількох цілей. Єдиний рравильний знаходиться в пакеті org.epochx.gp.model.

Якщо виконати цей клас консольний висновок повинен бути потік 3 колонки:

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 5.0 |  | OR(IF(OR(AND(OR(AND(D2 D3) AND(D2 D1)) ... |
| 1 | 5.0 |  | OR(IF(OR(AND(OR(AND(D2 D3) AND(D2 D1)) ... |
| 2 | 3.0 |  | IF(IF(NOT(IF(OR(IF(D3 D1 D3) NOT(D1)) ... |
| 3 | 3.0 |  | IF(IF(NOT(IF(OR(IF(D3 D1 D3) NOT(D1)) ... |
| 4 | 2.0 |  | IF(IF(NOT(IF(OR(IF(D3 D1 D3) NOT(D1)) ... |
| 5 | 2.0 |  | IF(IF(NOT(IF(OR(IF(D3 D1 D3) NOT(D1)) ... |
| 6 | 2.0 |  | IF(IF(NOT(IF(OR(IF(D3 D1 D3) NOT(D1)) ... |
| ... | ... |  | ... |

Перший стовпець - номер покоління, коли покоління 0 є результатом ініціалізації. Потім другі і треті стовпці є мінімальним фітнесом (тобто кращого фітнесу при використанні стандартизованої придатності) і найкраща програма, відповідно до рядків 5 і 6 нашого коду. Ви можете експериментувати з полями різної статистики, доступних в класі StatField.

Якщо все працює нормально, то ви повинні побачити, що мінімальне значення придатності поступово знижується до нуля в наступних поколіннях.

## **Налаштування параметрів**

Це мінімум, щоб запустити модель, і в даний час ми використовуємо всі стандартні значення, зазначені моделі і класи GPModel, але тепер ми хочемо надати нашим. Давайте встановимо розмір популяції, число поколінь і максимальну глибину програми.

1. GPModel model = new EvenParity(4);

2. model.setPopulationSize(500);

3. model.setNoGenerations(100);

4. model.setMaxDepth(8);

5. Life.get().addGenerationListener(new GenerationAdapter(){

6. public void onGenerationEnd() {

7. Stats.get().print(GEN\_NUMBER,

8. GEN\_FITNESS\_MIN,

9. GEN\_FITTEST\_PROGRAM);

10. }

11. });

12. model.run();

                       
 Рядки2, 3 і 4 були вставлені, щоб встановити ці параметри. Ще один цікавий параметр, який ми можемо встановити це число прогонів setNoRuns (INT), який робить виконуючу 50 або 100 пробігів для експериментів дуже легко.

## **Установка компонентів**

А також в якості основних параметрів прогону, багато які з компонентів в EpochX конфігуруються і міняються. Точно так, як ми встановлюємо параметри, ми можемо встановити ці компоненти. У наступному коді ми змінимо оператор кросовера, селектор програм і генератор випадкових чисел.

1. GPModel model = new EvenParity(4);

2. model.setCrossover(new UniformPointCrossover(model));

3. model.setProgramSelector(new TournamentSelector(model, 7));

4. model.setRNG(new MersenneTwisterFast());

5. Life.get().addGenerationListener(new GenerationAdapter(){

6. public void onGenerationEnd() {

7. Stats.get().printGenerationStats(GEN\_NUMBER,

8. GEN\_FITNESS\_MIN,

9. GEN\_FITTEST\_PROGRAM);

10. }

11. });

12. model.run();

Рядки 2, 3 і 4 були змінені, щоб встановити нові компоненти. Основна відмінність полягає в тому, що ми повинні передати в екземплярі компонента для використання, і він є загальним для компонентів, щоб мати доступ до моделі, яка використовується, так як він може визначити відповідні параметри для компонента. Деякі компоненти також мають свої власні варіанти, яке встановлюється з аргументами конструктора або сетер методів, таких як новий TournamentSelector, який приймає розмір турніру в якості аргументу.

## **Цілісний код та результати роботи**

import static org.epochx.stats.StatField.\*;

import org.epochx.gp.model.\*;

import org.epochx.gp.op.crossover.UniformPointCrossover;

import org.epochx.life.\*;

import org.epochx.op.selection.TournamentSelector;

import org.epochx.stats.Stats;

import org.epochx.tools.random.MersenneTwisterFast;

public class Example1 {

public static void main(String[] args) {

// Construct the model.

final GPModel model = new EvenParity(4);

// Set parameters.

model.setPopulationSize(500);

model.setNoGenerations(100);

model.setMaxDepth(8);

// Set operators and components.

model.setCrossover(new UniformPointCrossover(model));

model.setProgramSelector(new TournamentSelector(model, 7));

model.setRNG(new MersenneTwisterFast());

// Request statistics every generation.

Life.get().addGenerationListener(new GenerationAdapter(){

@Override

public void onGenerationEnd() {

Stats.get().print(GEN\_NUMBER, GEN\_FITNESS\_MIN, GEN\_FITTEST\_PROGRAM);

}

});

// Run the model.

model.run();

}

}

Результатом програми є логічна функція, що за заданими вхідними параметрами дає в результаті логічний 0. На рис. 7 наведено результат програми, при 22 популяціях ГА знайшов найпридатнішу функцію.

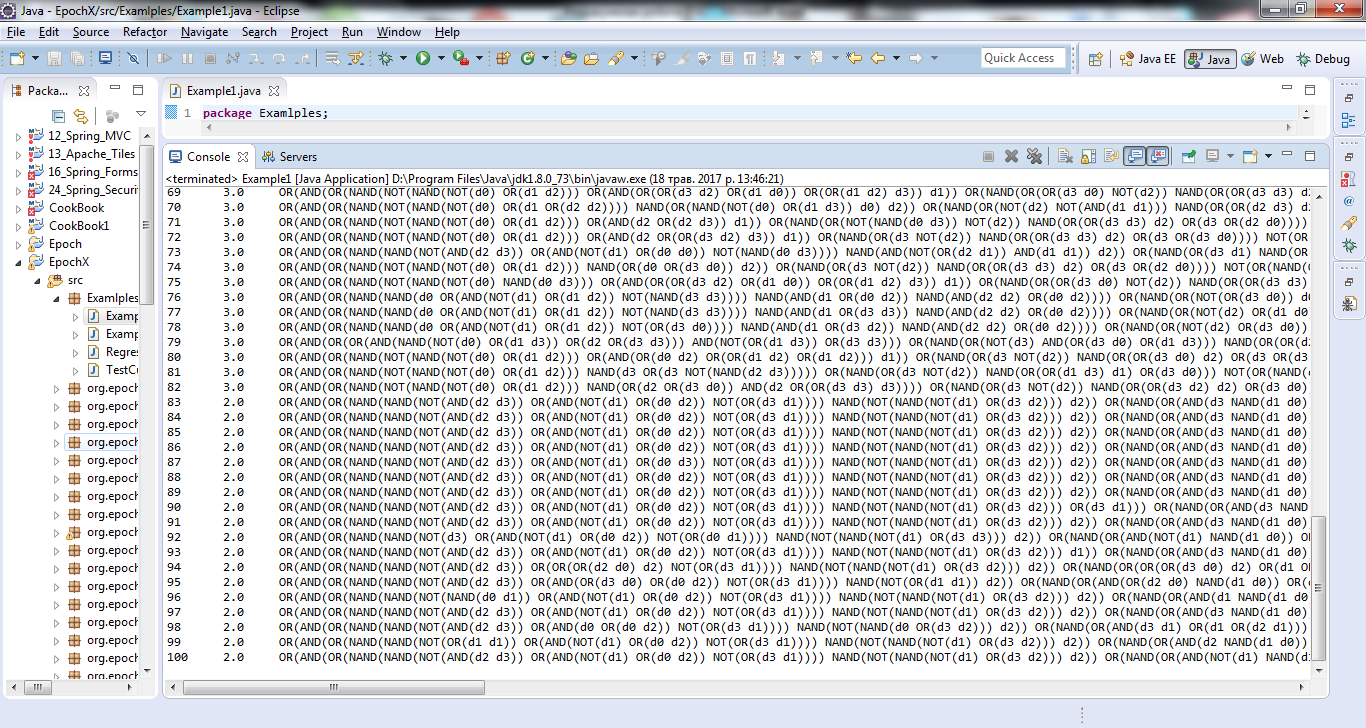


Рис. 7 Результат роботи ГА

**Приклад**

У той час як еталонні моделі, які забезпечені EpochX визначають багато з проблем, які ви можете захотіти виконати, безсумнівно, необхідно мати можливість створювати свої власні. Створення нової моделі в EpochX простий процес, хоча. 6-розрядний мультиплексор модель вже існує в вбудованих моделях EpochX, але для цілей демонстрації простої моделі, ми створимо нову.

Fitness в програмі буде 64 мінус число правильних виходів після спроби всіх 64 вхідних комбінацій, тому fitness 0.0 означатиме успіх на всі входи.

import java.util.\*;

import org.epochx.epox.\*;

import org.epochx.epox.bool.\*;

import org.epochx.gp.model.GPModel;

import org.epochx.gp.representation.\*;

import org.epochx.representation.CandidateProgram;

import org.epochx.tools.util.BoolUtils;

public class Mux6 extends GPModel {

private BooleanVariable d3;

private BooleanVariable d2;

private BooleanVariable d1;

private BooleanVariable d0;

private BooleanVariable a1;

private BooleanVariable a0;

// The boolean inputs/outputs that we will test solutions against.

private boolean[][] inputs;

private boolean[] outputs;

public Mux6() {

// Construct the variables into fields.

d3 = new BooleanVariable("d3");

d2 = new BooleanVariable("d2");

d1 = new BooleanVariable("d1");

d0 = new BooleanVariable("d0");

a1 = new BooleanVariable("a1");

a0 = new BooleanVariable("a0");

List<Node> syntax = new ArrayList<Node>();

// Functions.

syntax.add(new IfFunction());

syntax.add(new AndFunction());

syntax.add(new OrFunction());

syntax.add(new NotFunction());

// Terminals.

syntax.add(d3);

syntax.add(d2);

syntax.add(d1);

syntax.add(d0);

syntax.add(a1);

syntax.add(a0);

setSyntax(syntax);

// Generate set of test inputs and corresponding correct output.

inputs = BoolUtils.generateBoolSequences(6);

outputs = generateOutputs(inputs);

}

@Override

public double getFitness(CandidateProgram p) {

GPCandidateProgram program = (GPCandidateProgram) p;

double score = 0;

for (int i=0; i<inputs.length; i++) {

// Set the variables.

a0.setValue(inputs[i][0]);

a1.setValue(inputs[i][1]);

d0.setValue(inputs[i][2]);

d1.setValue(inputs[i][3]);

d2.setValue(inputs[i][4]);

d3.setValue(inputs[i][5]);

Boolean result = (Boolean) program.evaluate();

if (result == outputs[i]) {

score++;

}

}

return 64 - score;

}

/\*

\* Generates the correct outputs for the 6-bit multiplexer from

\* the given inputs to test against.

\*/

private boolean[] generateOutputs(boolean[][] in) {

boolean[] out = new boolean[in.length];

for (int i=0; i<in.length; i++) {

if(in[i][0] && in[i][1]) {

out[i] = in[i][2];

} else if(in[i][0] && !in[i][1]) {

out[i] = in[i][3];

} else if(!in[i][0] && in[i][1]) {

out[i] = in[i][4];

} else if(!in[i][0] && !in[i][1]) {

out[i] = in[i][5];

}

}

return out;

}

public static void main(String[] args) {

new Mux6().run();

}

}

# **Висновки**

В результаті виконання роботи було досліджено систему EpochX. EpochX є фреймворком Java для вивчення еволюції комп'ютерних програм, отриманих з використанням генетичних алгоритмів програмування. Фреймворк EpochX призначений в першу чергу для дослідників, що працюють в сфері по генетичній теорії програмування.

Даний фреймворк уже є доволі старим, до того ж остання стабільна версія вийшла аж у 2012 році. У свою чергу він також потребує доволі специфічні бібліотеки, що уже не входять в сучасні версії збірок Java. Однак він все ж є потужним інструментом і може коректно обробляти великі об’єми статистичних даних за допомогою розроблених в ньому потужних еволюційних алгоритмів.