ДОДАТОК А

Програмні коди

Інрефейси пакету diploma.library.api

```
package diploma.library.api;

public interface Data {
}

package diploma.library.api;

import java.util.List;

public interface Provider {
    List<Rule> getAll();
}
```

Інрефейси пакету diploma.engine.api

```
package diploma.engine.api;

public interface Rule {
    void add(Data data);
    Data run();
    Class getOutput();
    boolean isInput(Class clazz);
}

public interface Graph {
  void addEdge(int src, int dst);
  List<Integer>[] getAdjacency();
  List<Integer> getTopology();
  List<Integer> dfs(int start);
}

public interface Processor {
  List<Data> activateRules(List<Data> situation, Reactor reactor);
}

public interface Reactor {
  void loadLibrary(String ruleProviderClassName) throws Throwable;
  Reactor getOptimizedReactor(List<Data> situation);
  Graph getGraph();
  List<Rule> getRules();
}
```

Імплементація інтерфейсу Graph

```
package diploma.engine.impl;

import diploma.engine.api.Graph;

import java.util.*;

public class GraphImpl implements Graph {

  private int numberOfVertices;
  private List<Integer>[] adjacency;

  public GraphImpl(int numberOfVertices) {
    this.numberOfVertices = numberOfVertices;
    adjacency = new ArrayList[numberOfVertices];
    for (int i = 0; i < numberOfVertices; i++)  adjacency[i] = new
ArrayList<>();
  }

  @Override
  public void addEdge(int src, int dst) {
    adjacency[src].add(dst);
  }

  @Override
  public List<Integer>[] getAdjacency() {
    return adjacency;
  }

  @Override
  public List<Integer> getTopology() {
    int[] inDegree = new int[numberOfVertices];
    fillInDegreesOfVertices(inDegree);
    Queue<Integer> queue =
queueOfVerticesWithIndegreeZero(inDegree);
    return createGraphTopology(queue, inDegree);
  }

  private void fillInDegreesOfVertices(int[] inDegree) {
    for (int i = 0; i < numberOfVertices; i++)
      adjacency[i].forEach(node -> inDegree[node]++);
  }

  private Queue<Integer> queueOfVerticesWithIndegreeZero(int[]
indegree) {
    Queue<Integer> queue = new LinkedList<>();
    for (int i = 0; i < numberOfVertices; i++)
      if (indegree[i] == 0) queue.add(i);
    return queue;
  }
```

```
  private List<Integer> createGraphTopology(Queue<Integer> queue,
int[] inDegree) {
    int countOfVisitedVertices = 0;
    List<Integer> topology = new ArrayList<>();
    while (!queue.isEmpty()) {
      int u = queue.poll();
      topology.add(u);
      adjacency[u].forEach(
         node -> {
            if (--inDegree[node] == 0) queue.add(node);
          });
      countOfVisitedVertices++;
    }
    validateCycle(countOfVisitedVertices);
    return topology;
  }

  private void validateCycle(int countOfVisitedVertices) {
    if (countOfVisitedVertices != numberOfVertices)
      throw new IllegalArgumentException("There exists a cycle in
the graph!");
  }

  @Override
  public List<Integer> dfs(int start) {
    boolean[] visited = new boolean[numberOfVertices];
    List<Integer> dfsList = new ArrayList<>();
    dfsUtil(start, visited, dfsList);
    return dfsList;
  }

  void dfsUtil(int v, boolean visited[], List<Integer> dfsList) {
    visited[v] = true;
    dfsList.add(v);
    for (int n : adjacency[v]) {
      if (!visited[n]) dfsUtil(n, visited, dfsList);
    }
  }

  @Override
  public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    GraphImpl graph = (GraphImpl) o;
    return numberOfVertices == graph.numberOfVertices &&
Arrays.equals(adjacency, graph.adjacency);
  }

  @Override
  public int hashCode() {
    int result = Objects.hash(numberOfVertices);
    result = 31 * result + Arrays.hashCode(adjacency);
    return result;
```

```
  }

  @Override
  public String toString() {
    return "GraphImpl{"
        + "numberOfVertices="
        + numberOfVertices
        + ", adjacency="
        + Arrays.toString(adjacency)
        + '}';
  }
}
```

Імплементація інтерфейсу Processor

```
package diploma.engine.impl;

import diploma.engine.api.Processor;
import diploma.engine.api.Reactor;
import diploma.library.api.Data;
import diploma.library.api.Rule;

import java.util.ArrayList;
import java.util.List;

public class ProcessorImpl implements Processor {
    @Override
    public List<Data> activateRules(List<Data> situation, Reactor
reactor) {
        List<Data> result = new ArrayList<>();
        List<Rule> rules = reactor.getRules();
        List<Integer> topology = reactor.getGraph().getTopology();
        List<Integer>[] adjacency =
reactor.getGraph().getAdjacency();
        topology.forEach(
                index -> {
                    Rule rule = rules.get(index);
                    for (Data data : situation) {
                        if (rule.isInput(data.getClass())) {
                            rule.add(data);
                        }
                    }
                    Data outputData = rule.run();
                    if (adjacency[index].isEmpty()) {
                        result.add(outputData);
                    } else {
                        adjacency[index].forEach(
                                adjacencyIndex ->
rules.get(adjacencyIndex).add(outputData)
                        );
                    }
                }
        );
```

```
        return result;
    }
}
```

Імплементація інтерфейсу Reactor

```
package diploma.engine.impl;

import diploma.engine.api.Graph;
import diploma.engine.api.Reactor;
import diploma.library.api.Data;
import diploma.library.api.Provider;
import diploma.library.api.Rule;

import java.lang.reflect.Constructor;
import java.util.*;

public class ReactorImpl implements Reactor {

    private List<Rule> rules = new ArrayList<>();
    private Graph graph;

    public ReactorImpl() {
    }

    public ReactorImpl(List<Rule> rules) {
        setRules(rules);
    }

    @Override
    public void loadLibrary(String className) throws Throwable {
        Class<?> aClass = Class.forName(className);
        Constructor<?> constructor = aClass.getConstructor();
        Provider provider = (Provider) constructor.newInstance();
        this.rules.addAll(provider.getAll());
        graph = createGraphFromRules();
    }

    @Override
    public Reactor getOptimizedReactor(List<Data> situation) {
        Set<Integer> verticesSet = new HashSet<>();
        for (Data data : situation) {
            for (int i = 0; i < rules.size(); i++) {
                if (rules.get(i).isInput(data.getClass())) {
                    verticesSet.addAll(graph.dfs(i));
                }
            }
        }
        List<Integer> listOfVertices = new ArrayList<>(verticesSet);
        List<Rule> slimRuleList = new ArrayList<>();
        for (Integer ruleIndex : listOfVertices) {
            slimRuleList.add(rules.get(ruleIndex));
        }
```

```
    return new ReactorImpl(slimRuleList);
}

Graph createGraphFromRules() {
    int dimension = rules.size();
    Graph graph = new GraphImpl(dimension);
    for (int x = 0; x < dimension; x++) {
        Class outputClass = rules.get(x).getOutput();
        for (int y = 0; y < dimension; y++) {
            if (rules.get(y).isInput(outputClass)) {
                graph.addEdge(x, y);
            }
        }
    }
    return graph;
}

public List<Rule> getRules() {
    return rules;
}

public void setRules(List<Rule> rules) {
    this.rules = rules;
    graph = createGraphFromRules();
}

public Graph getGraph() {
    return graph;
}

public void setGraph(Graph graph) {
    this.graph = graph;
}

@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    ReactorImpl reactor = (ReactorImpl) o;
    return Objects.equals(rules, reactor.rules) &&
            Objects.equals(graph, reactor.graph);
}

@Override
public int hashCode() {
    return Objects.hash(rules, graph);
}

@Override
public String toString() {
    return "ReactorImpl{" +
            "rules=" + rules +
            ", graph=" + graph +
```

```
                '}';
    }
}
```

Імплементація інтерфейсу Data

```
package diploma.benchmark.model;

import diploma.library.api.Data;
import java.util.Objects;

public class FactA implements Data {

  private boolean valid;

  public FactA(boolean valid) {
    this.valid = valid;
  }

  public boolean isValid() {
    return valid;
  }

  public void setValid(boolean valid) {
    this.valid = valid;
  }

  @Override
  public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    FactA that = (FactA) o;
    return valid == that.valid;
  }

  @Override
  public int hashCode() {
    return Objects.hash(valid);
  }

  @Override
  public String toString() {
    return "FactA{" + "valid=" + valid + '}';
  }
}
```

Приклади реалізації бенчмарку для Drools

```
package benchmark.drools;

import diploma.benchmark.model.FactI;
import org.kie.api.KieServices;
import org.kie.api.runtime.KieContainer;
```

```
import org.kie.api.runtime.KieSession;
import org.openjdk.jmh.annotations.*;

@State(Scope.Benchmark)
public class BenchMarkA {
  private KieServices ks = KieServices.Factory.get();
  private KieContainer kContainer = ks.getKieClasspathContainer();
  private FactI fact = new FactI(true);

  @Fork(value = 1, warmups = 1)
  @Benchmark
  @BenchmarkMode(Mode.Throughput)
  @Warmup(iterations = 5)
  @Measurement(iterations = 10)
  public void act_01() {
    KieSession kSession = kContainer.newKieSession("ksession-
rules");
    kSession.insert(fact);
    kSession.fireAllRules();
    kSession.destroy();
  }
}
```

Приклади правила для Drools

```
package rules;
dialect  "mvel"

import diploma.benchmark.model.InputFact
import diploma.benchmark.model.FactA

rule "RuleA"
    when
    i : InputFact(valid == true)
    then
    insert(new FactA(true));
end
```

Приклади реалізації бенчмарку для EasyRools

```
package benchmark.easyrules;

import benchmark.easyrules.rules.RuleC;
import diploma.benchmark.model.FactB;
import org.jeasy.rules.api.;
import org.jeasy.rules.api.Rules;
import org.jeasy.rules.api.RulesEngine;
import org.jeasy.rules.core.RulesEngineBuilder;
import org.openjdk.jmh.annotations.*;

@State(Scope.Benchmark)
public class BenchMarkA {
```

```
  private Facts facts = new Facts();
  private Rules rules = new Rules();
  private RulesEngine rulesEngine =

RulesEngineBuilder.aNewRulesEngine().withSilentMode(true).build();

  @Setup(Level.Invocation)
  public void setUp() {
    facts.put("FactB", new FactB(true));
    rules.register(new RuleC());
  }

  @Fork(value = 1, warmups = 1)
  @Benchmark
  @BenchmarkMode(Mode.Throughput)
  @Warmup(iterations = 5)
  @Measurement(iterations = 10)
  public void oneRule() {
    rulesEngine.fire(rules, facts);
  }
}
```

Приклади правила для EasyRools

```
package benchmark.easyrules.rules;

import diploma.benchmark.model.FactA;
import diploma.benchmark.model.InputFact;
import org.jeasy.rules.annotation.*;
import org.jeasy.rules.api.Facts;

@Rule(name = "RuleA", description = "Benchmark rule number one.")
public class RuleA {

  @Condition
  public boolean when(@Fact("InputFact") InputFact model) {
    return model.isValid();
  }

  @Action
  public void then(Facts facts) throws Exception {
    facts.remove("InputFact");
    facts.put("FactA", new FactA(true));
  }

  @Priority
  public int getPriority() {
    return 1;
  }
}
```

ДОДАТОК Б

Слайди презентації

Міністерство освіти і науки України
Харківський національний інститут радіоелектроніки

Атестаційна робота магістра

Дослідження та розробка моделі знаходження рішень в задачах з невизначеним кінцевим станом

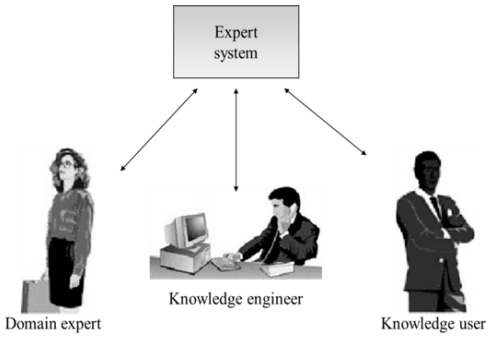
Науковий Керівник:
д.т.н., проф. **Четвериков Г.Г.**

Виконав:
Студент групи ІПЗмзд-17-1 **Вакарь Л.Г.**

1

Експертна система

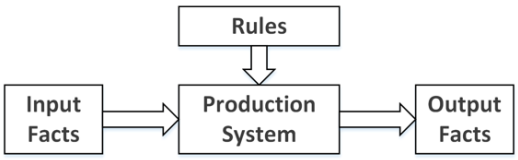
Умовно всі задачі можна розділити на два типи. Задачі з заданим кінцевим станом і задачі з не заданим кінцевим станом. Для задач з заданим кінцевим станом характерний пошук шляху досягнення кінцевого стану. Задачі з не заданим кінцевим станом повинні прийти до кінцевого стану в ході процесу їх розв'язання. У таких завданнях результат не детермінований, певним і відомим є тільки початкова ситуація і набір методів її розв'язання. Розв'язанням таких задач займається різновид систем штучного інтелекту, а саме експертні системи.



Експертна система – це комп'ютерна програма, здатна імітувати поведінку людського експерта у вузькій галузі знань.

2

Двигун висновків експертної системи



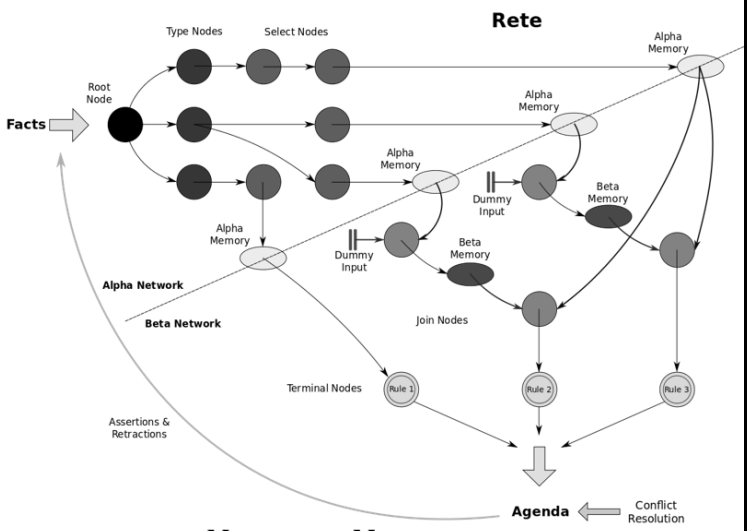
ЯКЩО
зарплата більше 10 000
І
зарплата менше 20 000
ТО
податок 10%

Експертна система складається з кількох компонентів: бази знань та двигуна висновків. База знань це сукупність правил збережених у форматі “якщо-то”. Двигун висновків витягує правила із бази знань і, якщо відомі факти відповідають предикатам правила, застосовує його для виведення нових фактів. Для виконання правил та отримання нових фактів двигуни висновків використовують алгоритми продукційних систем.

3

Rete алгоритм

Сучасні експертні системи використовують двигуни висновків створені на основі імплементацій rete-алгоритму. Rete-алгоритм становить собою дуже швидкий засіб зіставлення фактів з предикатами правил. Висока швидкодія досягається шляхом зберігання в оперативній пам'яті інформації про правила представлені у вигляді мережі. Rete-алгоритм жертвує великою кількістю пам’яті задля досягнення великої швидкості виконання. Для його реалізації повинні бути створені не тільки об’єкти правил та моделей, але об’єкти предикатів та об’єкти полів моделей.



$$N_{obj} = N_{rules} + N_{predicates\_of\_rules} + N_{facts} + N_{fields\_of\_facts}$$

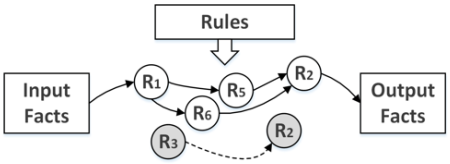
4

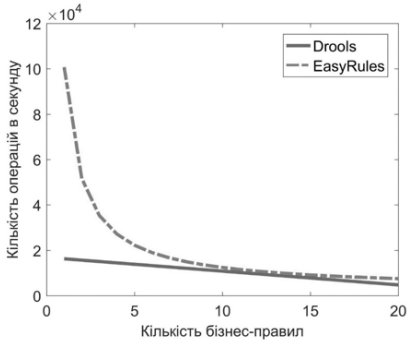
Мета і напрямок дослідження

Метою роботи є розробка моделі поліпшеного двигуна висновків експертних систем. Модель повинна використовувати менше оперативної пам'яті та не поступатися у швидкості наявним аналогам.

Зменшити кількість використовуваної пам'яті можна шляхом знаходження необхідних для прийняття експертного рішення правил без побудови rete мережі.

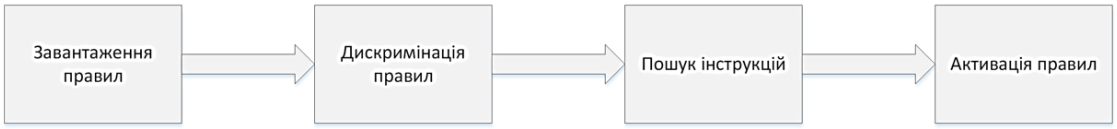
Порівняльне дослідження систем Drools та EasyRules виявили, що система EasyRules працює швидше шляхом використання правил написаних мовою програмування Java. Цей підхід може потенційно збільшити швидкість розроблюваного двигуна.





5

Новий алгоритм продукційної системи



Алгоритм роботи двигуна висновків повинен складатися з декількох функціональних компоненті, які повинні забезпечити максимальну ефективність роботи двигуна висновків. Функціональними елементами алгоритму є:

1) завантаження правил та побудова спрямованого ациклічного графа;
2) дискримінація непотрібних правил;
3) побудова інструкцій виконання;
4) запуск правил.

6

Структура правила

Правило має простий алгоритм роботи. Спочатку відбувається накопичення необхідних для роботи правила фактів в масив *WM*. Коли в наявності є всі необхідні факти відбувається процес виконання правила. Правило тестує *WM* на відповідність предикату *P(WM)*, якщо предикат повертає істину на об'єктах *WM* виконується функція *F(WM)*, яка продукує *rf*. У іншому випадку правило продукує модель за замовчуванням rf_d. Загальна форма алгоритму роботи правила може бути представлена формулою (8).

$$P(WM)\begin{cases} false \rightarrow F(WM) \rightarrow rf_d \\ true \rightarrow F(WM) \rightarrow rf \end{cases}$$

Факти типу *rf* можуть входити до *WS* правил, це дозволяє побудувати мережу правил пов'язаних між собою зв'язками вхідних та вихідних фактів. Факт який продукує правило не може бути частиною його *WM*. Така заборона дозволяє для заданого набору правил *R*, побудувати направлений ациклічний граф *G*.

```
public class RuleA implements Rule {

 private Data[] dataArray = new Data[1];

 @Override
 public void add(Data data) { dataArray[0] = data; }

 @Override
 public Data run() {
  InputFact inputFact = (InputFact) dataArray[0];
  return inputFact.isValid() ? new FactA(true) : new FactA(false);
 }

 @Override
 public Class getOutput() { return FactA.class; }

 @Override
 public boolean isInput(Class clazz) {
  return clazz.isAssignableFrom(InputFact.class);
 }
}
```

7

Завантаження правил та створення графа

База знань для алгоритму представлена у вигляді правил написаних безпосередньо мовою програмування. До алгоритму правила надає провайдер у вигляді масиву (alg. 1). Складність алгоритму становить:

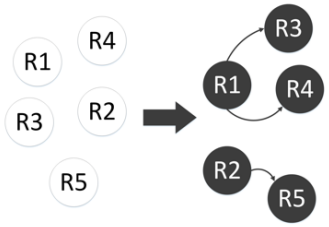
$$O(R), \text{де } R \text{ — кількість правил}$$

З масиву правил створюється спрямований ациклічний граф у вигляді масиву суміжних вершин. При цьому процесі кожне правило перевіряється на зв'язок між вхідними та вихідними об'єктами. У разі наявності такого зв'язку додається суміжна вершина (alg. 2). Складність алгоритму становить:

$$O(R^2), \text{де } R \text{ — кількість правил}$$

```
ALGORITHM #1
1.    while rules exists
2.          create object of rule
3.          add rule object to R
4.    return R
```

```
ALGORITHM #2
1.    Procedure(R)
2.    initialize G
3.    for each ruleX in R
4.          for each ruleY in R
5.                if ruleX.rf in ruleY.WM then
6.                      G add [ruleX, ruleY]
7.    return G
```



8

Дискримінація правил

Якщо бібліотека буде містити десятки тисяч правил *R*, а для обробки запиту та формування експертного висновку потрібно лише сто з них, то розгляд і намагання виконати непотрібні правила, це втрата часу та комп'ютерних ресурсів.

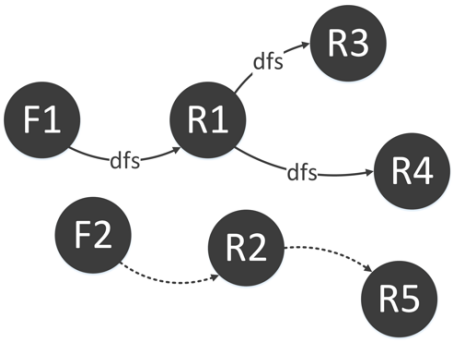
Коли до моделі надходить ситуація *S*. Модель двигуна висновків використовує ситуацію для знаходження тільки тих правил які можуть бути використані для її обробки. Для цього можна буде використати алгоритми пошуку вглибину графа (dfs). Пошук в глибину графа потрібно провести для кожного факту з множини *S*. Результати пошуку потрібних правил зберігаються у набір в якому немає дублікатів правил R_d. Складність алгоритму становить:

$$O(S(R+E)), \text{де}$$

S – кількість фактів ситуації

R – кількість правил в графі

E – кількість ребер між вершинами графу




```
ALGORITHM #3
1.    Procedure(S, G)
2.    init  Rd
3.    for each sf in S
4.          Rd add ← dfs(G, sf)
5.    return  Rd
```

9

Знаходження інструкцій виконання

Для знаходження порядку обробки правил потрібно побудувати *топології графа*. Одним з можливих алгоритмів побудови топології може слугувати алгоритм Кана. Крім топології графа у вигляді інструкції використовується *мапа суміжних вершин*, вона вказує на всі вершини які можна досягти з заданої. Цю мапу використовує останній алгоритм для знаходження експертного висновку. Результатом роботи алгоритму (alg. 4) є топологія графа та мапа суміжних вершин. Складність алгоритму становить:

$O(R_d^2)$, де R_d – кількість відібраних правил


```
ALGORITHM #4
1.    Procedure(Rd)
2.    init Graph
3.    for each ruleX in Rd
4.          for each ruleY in Rd
5.                if x.rf in y.WM then
6.                      Graph add [x, y]
7.    Topology ← Graph.getTopology()
8.    Adjacency ← Graph.getAdjacency()
9.    return [Topology, Adjacency]
```

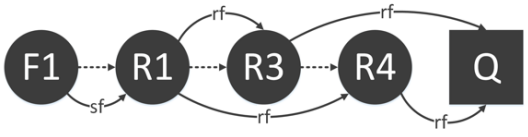
10

Виконання правил

Алгоритм (alg. 5) приймає на вхід масив правила (R_d), топологію графа (Topology), мапу суміжних вершин (Adjacency) та ситуацію (S). Кожне правило активується в порядку описаному в топології графа. Якщо правило потребує фактів ситуації як вхідний, ситуація додається до правила (лінії 5–7) алгоритму. Потім правило активується. За допомогою мапи суміжних вершин продукт роботи правила додається до робочої пам'яті *WM* інших правил. У випадку якщо правило не має суміжних вершин його продукт додається до масиву результатів *Q*. Складність алгоритму становить:

$$O(R_d + R_d^2), \text{де}$$

R_d – кількість відібраних правил

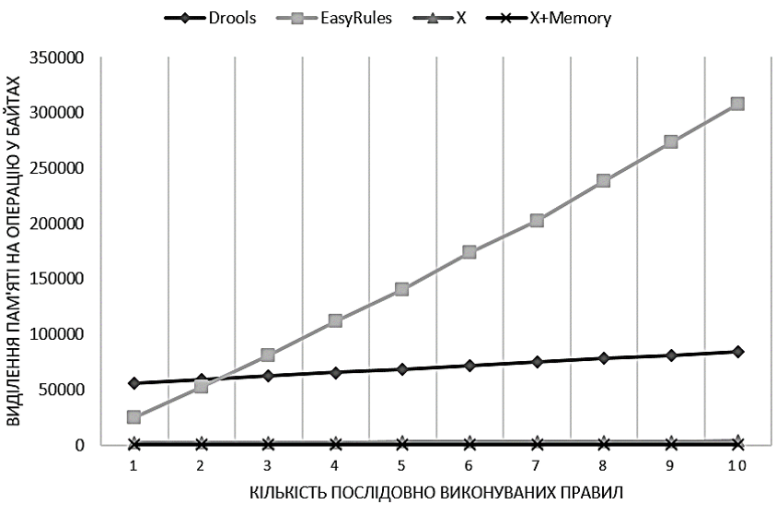



```
ALGORITHM #5
1.    Procedure(Rd, S, Topology, Adjacency)
2.    init Q
3.    for each index in topology
4.          r ← Rd[index]
5.          for each sf in S
6.                if sf in r.WM then
7.                      r.WM add sf
8.          rf ← r.run()
9.          Vertex_Adjacency ← Adjacency[index]
10.         if Vertex_Adjacency not empty
11.               for each i in vertex_adjacency
12.                     forwardRule ← Rd[i]
13.                     forwardRule.WM add rf
14.         else
15.               Q add rf
16.   return Q
```

11

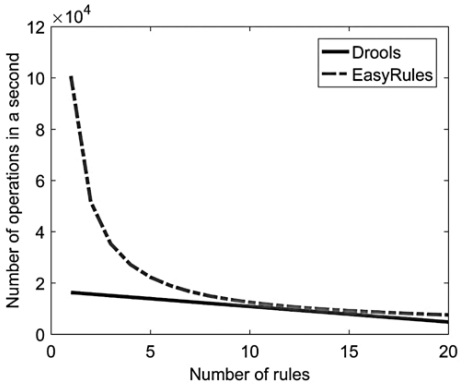
Результати тестування використання пам'яті

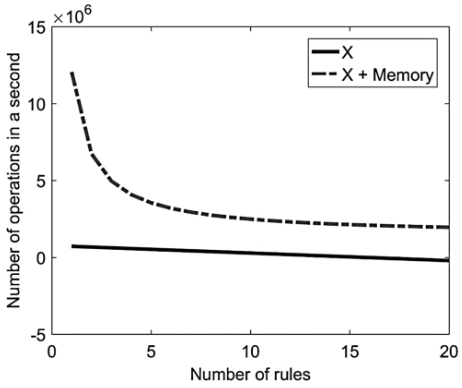
Результати тестування демонструють лінійну залежність виділення пам'яті від кількості виконуваних правил. За цим критерієм найгіршим виявився двигун висновків EasyRules. Набагато ліпшим за нього виявився двигун висновків Drools. Розроблена модель показала найліпший результат. Порівнюючи двигуни висновків Drools та EasyRules можна сказати, що останній втрачає переваги вже після виконання двох послідовних правил. У порівнянні з ними розроблена модель двигуна висновків X демонструє перевагу у декілька порядків.



12

Результати тестування швидкодії





$y_{Drools} = 16865{,}20 - 605{,}36x$

$y_{\text{EasyRools}} = 2606{,}41 + 98093{,}22/x$

$y_X = 760469.73 - 48856.93x$

$y_{X+memory} = 1415242.92 + 10626595.48/x$

Виходячи з результатів порівняльного дослідження найгірший результат показав двигун висновків Drools. Не дивлячись на схожість математичних моделей Drools та розробленої моделі без використання пам'яті, остання набагато швидша, та менш залежна від кількості правил. При використанні пам'яті в розробленій моделі її результат значно покращується. При цьому багаторазове пришвидшення спостерігається при кількості правил від одного до п'яти.

13

Можливості застосування

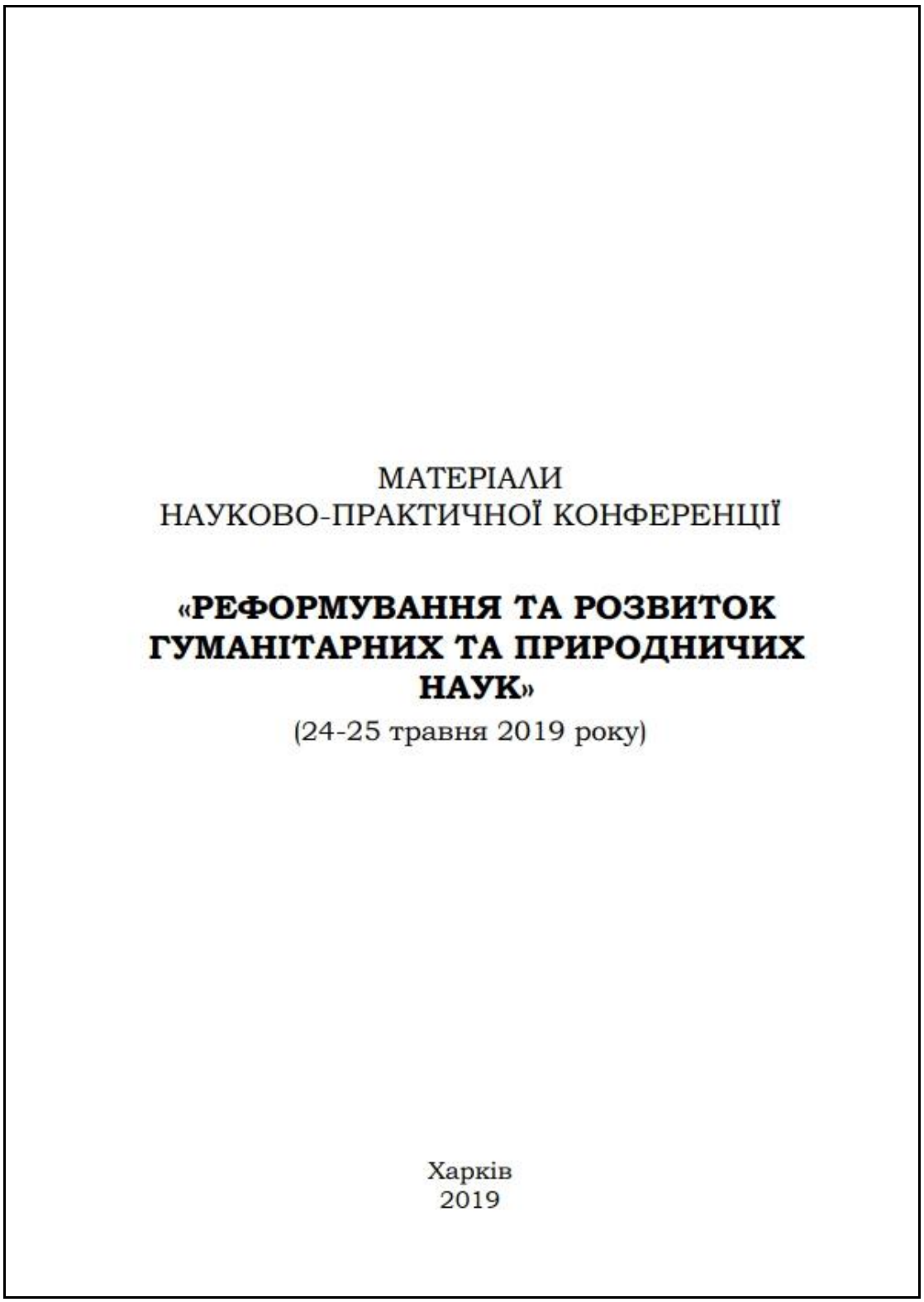
Розроблена модель продукційної системи має дві головні переваги: швидкість роботи та невелике споживання оперативної пам'яті. Завдяки швидкості роботи двигуни висновків створені на основі розробленої моделі можуть заміни наявні системи.

Також вони можуть бути використані для побудови високо навантажених мережевих систем. Перевага в використанні значно меншої кількості оперативної пам'яті може значно розширити сферу використання двигунів висновків. Вони зможуть виступати компонентом для мобільних додатків, або програмного забезпечення розумних пристроїв.

14

ДОДАТОК В

Тези конференції «Реформування та розвиток гуманітарних та природничих наук»


МАТЕРІАЛИ
НАУКОВО-ПРАКТИЧНОЇ КОНФЕРЕНЦІЇ

«РЕФОРМУВАННЯ ТА РОЗВИТОК ГУМАНІТАРНИХ ТА ПРИРОДНИЧИХ НАУК»

(24-25 травня 2019 року)

Харків
2019

Рисунок В.1 – Титульний аркуш

УДК 001.38:009+50(063)
Р45

Реформування та розвиток гуманітарних та природничих наук.
Р45 Матеріали науково-практичної конференції (м. Харків, 24-25 травня 2019 року). – Херсон : Видавництво «Молодий вчений», 2019. – 168 с.

ISBN 978-617-7640-56-0

У збірнику представлені матеріали науково-практичної конференції «Реформування та розвиток гуманітарних та природничих наук». Розглядаються загальні питання архітектури та мистецтвознавства, біологічних, військових, географічних наук, державного управління, культурології та інші.

Збірник призначений для науковців, викладачів, аспірантів та студентів, а також для широкого кола читачів.

УДК 001.38:009+50(063)

ISBN 978-617-7640-56-0

Рисунок В.2 – Сторінка друкарні

ПОЛІТИЧНІ НАУКИ

ПСИХОЛОГІЧНІ НАУКИ

СОЦІАЛЬНІ КОМУНІКАЦІЇ

ТЕХНІЧНІ НАУКИ

Рисунок В.3 – Зміст

Список використаних джерел:

1. Реута О.В. Порівняння матричної і воксельної моделей тривимірного тіла для задач його реконструкції // Прикладна геометрія та інженерна графіка. – К.: КНУБА, 2009. – Вип. 82. – С. 203–207.
2. Реута О.В. Використання напівпрозорих вокселів в дискретних моделях просторових об'єктів // Праці Таврійського державного агротехнічного університету. Прикладна геометрія та інженерна графіка. – Мелітополь: ТДАТУ, 2012. – Вип. 4, т. 52. – С. 107–111.

Вакарь Л.Г.

студент,

Харківський національний університет радіоелектроніки

КОНЦЕПЦІЯ ПОБУДОВИ ПРОДУКЦІЙНОЇ СИСТЕМИ ДЛЯ ДВИГУНА ВИСНОВКІВ ЕКСПЕРТНИХ СИСТЕМ

В сучасному технічно складному світі, експерти відіграють дуже важливу роль. Вони є незамінним елементом в структурі багатьох організацій. Проте людські експерти мають ряд недоліків: суб'єктивне судження, обмеження швидкості прийняття рішень, невеликий час роботи на протязі доби тощо. Тому останнім часом, в галузях де прийняття рішень можливе на основі набору відомих фактів без особистої участі експерта, активно впроваджуються експертні системи. Експертна система – це комп'ютерна програма, здатна імітувати поведінку людського експерта у вузькій галузі знань [1, с. 52]. Наприклад, експертна система CaDet здатна розпізнавати рак на ранній стадії.

Експертна система складається з кількох компонентів: бази знань, двигуна висновків, модулю придбання знань, інтерфейсу пояснення [2]. База знань це сукупність правил збережених у форматі "якщо-то". Двигун висновків (рис. 1) витягує правила із бази знань і, якщо відомі факти відповідають предикатам правила, застосовує його для виведення нових фактів.

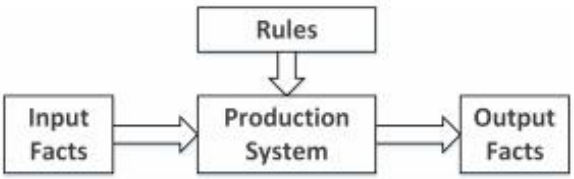


Рис. 1. Схема роботи двигуна висновків

Для виконання правил та отримання нових фактів двигуни висновків використовують алгоритми продукційних систем (ПС). Сучасні експертні системи використовують двигуни висновків створені на основі імплементацій rete-алгоритму. Rete-алгоритм становить собою дуже швидкий засіб зіставлення фактів з предикатами правил. Висока швидкодія досягається шляхом зберігання

Рисунок В.4 – Сторінка тез конференції

в оперативній пам'яті інформації про правила представлені у вигляді мережі [3, с. 83]. Правила для rete-алгоритму повинні бути написані з використанням спеціально розробленого синтаксису. Rete-алгоритм дозволяє виконувати двигуну висновків тільки ті правила які можуть бути виконані виходячи з наданих вхідних фактів.

Попри свою ефективність, моделі rete-алгоритму базуються на правилах, синтаксис яких не може охопити складну бізнес-логіку. На даний момент це питання вирішується додаванням до двигуна висновків додаткового шару бізнес-логіки [4]. Цього можна уникнути якщо представити правила у вигляді класів об'єктно-орієнтованої мови програмування (ООМП). Такі правила можуть мати безпосередній доступ до API додатку в якому використовується двигун висновків. На жаль rete-алгоритм не може бути застосований до правил написаних на ООМП через складність та непередбачуваності синтаксису. Двигуни висновків які здатні обробляти такі правила, наприклад EasyRules, використовують алгоритм Маркова як основу ПС. Алгоритм Маркова – це упорядкована група правил, які застосовуються в порядку пріоритету [3, с. 81]. На відміну від rete-алгоритму, алгоритм Маркова вимушений тестувати більше правил за необхідну, для прийняття експертного рішення, кількість. Це впливає на швидкість роботи двигуна висновків [3, с. 82].

У зв'язку з цим є актуальною розробка концепції ПС яка б поєднувала в собі можливість використовувати в правилах складну бізнес-логіку та ефективно обирати тільки необхідні для експертного рішення правила. Концепція ПС повинна поєднати дві базові ідеї. Перша це представлення правил у вигляді класів ООМП. Друга – використання ідей реалізованих в rete-алгоритмі для зменшення кількості правил які можуть бути використані під час процесу отримання експертного рішення. Ефективність роботи rete-алгоритму забезпечує представлення наборів предикатів правил у вигляді спрямованого ациклічного графа (САГ) на кінці якого знаходяться елементи виводу нових фактів [5]. Така концепція алгоритму дозволяє швидко знаходити правила, які можна застосувати до набору вхідних фактів проходячи до них крізь мережу графу.

Хоча синтаксис ООМП і не дозволяє побудувати мережу з предикатів правил, можна побудувати мережу із самих правил. Якщо мережа правил буде представлена у формі САГ виникне можливість використання властивостей графу для знаходження необхідних для виконання правил. Наприклад, для цього може бути використаний алгоритм пошуку в глибину [6, с. 26]. Для реалізації такого підходу концепція ПС повинна обробляти особливу структуру правила написаного на ООМП. Правило повинно складатися з масиву вхідних фактів, набору предикатів та функції. Функція виводить новий факт із масиву вхідних фактів, якщо ті відповідають набору предикатів. Факт який виводить одне правило може бути використаний іншим правилом. Це дозволяє побудувати мережу правил пов'язаних між собою зв'язками вхідних та вихідних фактів. Факт який продукує правило не може бути використаний самим правилом. Така заборона дозволяє, для заданого набору правил,

Рисунок В.5 – Сторінка тез конференції

побудувати САГ. При цьому кожне правило може бути використане тільки один раз.

Концепція роботи ПС виглядає наступним чином. При старті ПС завантажує масив правил з якого створюється САГ. Після цього ПС готова приймати запити представлені у вигляді масиву вхідних фактів. Обробка вхідних фактів та отримання експертного висновку здійснюється у два етапи. На першому етапі ПС використовує масив вхідних фактів для знаходження тільки тих правил, які можуть бути використані для їх обробки. Знайдені, для поточного набору класів вхідних фактів, правила зберігаються для повторного використання. Якщо до ПС вдруге надійде набір фактів представлений тим самим набором класів ООМП, вона використає збережений набір правил. На другому етапі ПС виконує обрані правила на наборі вхідних фактів та зберігає кінцеві факти в масив результатів. На схемі роботи ПС (рис. 2) показано процес виконання обраних правил. При цьому ПС оминає правила які не можуть бути виконані на вхідному наборі фактів.

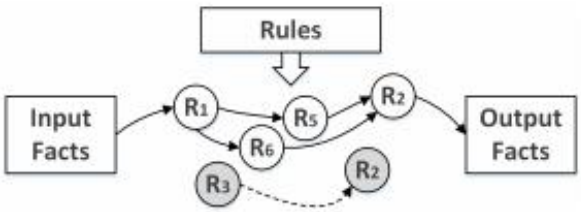


Рис. 2. Схема роботи продукційної системи

Таким чином в даному дослідженні запропонована концепція ПС, яка здатна використовувати правила зі складною бізнес-логікою та водночас обирати для виконання тільки ті правила які можуть бути застосовані виходячи із вхідного набору фактів. Експертні системи створені на основі запропонованої концепції зможуть швидше обробляти запити користувачів в складних, бізнес орієнтованих додатках.

Список використаних джерел:

1. Negnevitsky M. Artificial Intelligence: A Guide to Intelligent Systems. Second ed. // Harlow, 2005. – 415 p.

2. Forsyth R. The Architecture of Expert Systems. // Expert Systems: Principles & Case Studies. Ed. Forsyth R. / London, 1984. P. 9–17.

3. Джарратано Д., Райли Г. Экспертные системы: принципы разработки и програмирование, 4-е издание. – Москва, 2007. – 1152 с.

4. Poblete R.E., Fuente, D.D., Alonso M. Using Cloud Computing with RETE Algorithms in a Platform as a Service (PaaS) for Business Systems Development. // Proceedings of the 2011 International Conference on Artificial Intelligence / Las Vegas, Vol. 2, 2011. P. 654–658.

5. How the Rete Algorithm Works? URL: https://www.sparklinglogic.com/rete-algorithm-demystified-part-2/ (дата звернення: 10.05.2019).

6. Bang-Jensen J., Gutin G. Digraphs: Theory, Algorithms and Applications, Second ed. // London, 2008. – 795 p.

Рисунок В.6 – Сторінка тез конференції

ДОДАТОК Г

Тези конференції «SCIENCE, RESEARCH, DEVELOPMENT #17»

MONOGRAFIA
POKONFERENCYJNA

SCIENCE,
RESEARCH, DEVELOPMENT #17

TECHNICS AND TECHNOLOGY.

Belgrade (Serbia)

30.05.2019- 31.05.2019

Рисунок Г.1 – Титульний аркуш

U.D.C. 330+339.138+658+657+336.71+339+082
B.B.C. 94
Z 40

Zbiór artykułów naukowych recenzowanych.
(1) Z 40 Zbiór artykułów naukowych z Konferencji Miedzynarodowej Naukowo-Praktycznej (on-line) zorganizowanej dla pracowników naukowych uczelni, jednostek naukowo-badawczych oraz badawczych z państw obszaru byłego Związku Radzieckiego oraz byłej Jugosławii.

(*30.05.2019*) - Warszawa, 2019. - 84 str.
ISBN: 978-83-66401-02-0

Wydawca: Sp. z o.o. «Diamond trading tour»
Adres wydawcy i redakcji: 00-728 Warszawa, ul. S. Kierbedzia, 4 lok.103
e-mail: info@conferenc.pl

W artykułach naukowych zachowano oryginalną pisownię.

Wszystkie artykuły naukowe są recenzowane przez dwóch członków Komitetu Naukowego.

W przypadku cytowań obowiązkowe jest odniesienie się do monografii.

Nakład: 80 egz.

ISBN: 978-83-66401-02-0

Рисунок Г.2 – Сторінка друкарні

Рисунок Г.3 – Сторінка редакційного комітету

SPIS/СОДЕРЖАНИЕ

Рисунок Г.4 – Сторінка змісту

ПОРІВНЯЛЬНЕ ДОСЛІДЖЕННЯ ДВИГУНІВ ВИСНОВКІВ ЕКСПЕРТНИХ СИСТЕМ

Вакарь Л.Г.
студент шостого курсу
Харківський національний університет радіоелектроніки

Четвериков Г. Г.
доктор технічних наук, професор
Харківський національний університет радіоелектроніки

Ключові слова: експертна система, двигун висновків, система на основі правил, порівняльне дослідження, тест продуктивності.

Keywords: expert system, inference engine, rule-based system, comparative study, performance test.

У сучасному світі, для здійснення автоматизації та стандартизації оперативної діяльності, компанії використовують корпоративні додатки. В основі корпоративних додатків знаходяться дані та логіка їх обробки. Продовж певного часу для створення корпоративних додатків використовувалася програмна концепція "Дані + Алгоритм = Програма". У даній концепції за обробку даних відповідають алгоритми. Сукупність алгоритмів утворюють програмне рішення. В наш час все частіше для створення корпоративних додатків використовується інша концепція "Знання + Висновки = Система". Заснований на знаннях підхід до проектування систем це еволюційна зміна з революційними наслідками. Він замінює програмну традицію новою архітектурою, яка заснована на базі знань та механізмі висновків. Не дивлячись на схожість підходів вони відрізняються достатньо, щоб це мало глибокі наслідки [1].

База знань містить бізнес-правила які визначають, як програма обробляє дані. Бізнес-правила представлені у форматі "якщо-то". Вони можуть бути легко змінені, без втручання в код додатку. Бізнес-правила обробляє механізм висновків, або як його ще називають двигун висновків. Двигун висновків витягує правила із бази знань і, якщо відомі факти відповідають предикатам бізнес-правила, застосовує його для виведення нових фактів. Двигуни висновків засновані на алгоритмах продукційних систем. Сучасні системи бізнес правил використовують швидкий алгоритм зіставлення фактів з предикатами бізнес-правил – rete [2, с. 83]. Таким чином система розділена на два компоненти: база знань і двигун висновків. На даний момент на ринку існує велика кількість двигунів висновків: Drools, Oracle Policy Automation, IBM Operational Decision Manager on Cloud, Blaze Advisor тощо. Однак всі ці рішення або використовують велику кількість апаратних ресурсів і повинні бути розгорнуті на окремому сервері, або становлять собою комерційні хмарні сервіси.

Рисунок Г.5 – Сторінка тез конференції

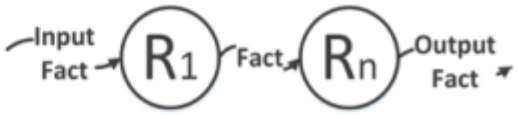


Рисунок 1 – Схема роботи правил

Малі та середні компанії, що б ефективно конкурувати з великими корпораціями, також змушені впроваджувати автоматизацію бізнес-процесів. Часто такі компанії обмежені у фінансових і апаратних ресурсах і не можуть, з тих чи інших причин, дозволити собі використовувати комерційні або ресурсомісткі системи керування бізнес-правилами. При цьому їм потрібні рішення, які не поступаються за швидкодією складним комерційним системам. Одним з варіантів такого рішення є бібліотека написана мовою програмування Java – EasyRules. Розмір бібліотеки EasyRules версії 3.0.0 складає всього 28 КВ. Для порівняння розмір бібліотеки Drools Core 7.0.0 складає 3,6 МВ. Drools використовує алгоритм rete [3]. EasyRules використовує правила, написані на синтаксисі Java. Швидкий алгоритм rete не може бути застосований до таких правил через складність синтаксису, тому EasyRules використовує менш ефективний алгоритм Маркова як основу продукційної системи.

Порівняльні дослідження продуктивності проводилися або для безкоштовних ресурсомістких продуктів [4, 5], або для комерційних систем [6]. У зв'язку з цим є актуальним порівняльне дослідження швидкодії ресурсомісткої системи Drools з некомерційним і компактним рішення EasyRules. Результати дослідження можуть бути використані, розробниками програмного забезпечення, для прийняття рішень щодо вибору системи керування бізнес-правилами при розробці програмних продуктів.

В ході дослідження було проведено серію однакових тестів для кожного з продуктів. Тести запускали послідовне виконання різної кількості правил, від 1-го до 10-ти. Кожне правило виконуючись виробляло певний факт. Цей факт використовувався наступним правилом для здобуття нового факту (рис.1).

Для проведення тестування використовувалися наступні апаратні ресурси: процесор Intel Core i7-8550U 4-cores 8-threads 4GHz, оперативна пам'ять DDR4 – 2133 МГц 16GB. Програмне середовище складалося з операційної систему ОС Linux Mint 19.1 tessa з ядром Linux Kernel 4.15.0-48-generic і версію JDK openjdk version "11.0.2" 2019-01-15. Для проведення вимірювань була використана бібліотека JMH. Був обраний тип вимірювання – кількість операцій в секунду. Вимірювання проводилися в рамках однієї сесії яка включала: попередній розігрів JVM 5 разів, отримання середнього значення за результатами 20 вимірювань. Результати тестування (табл.1) дозволили накопичити достатньо даних для проведення математичного моделювання.

Математична модель повинна показати залежність швидкості роботи двигуна висновків від кількості послідовно виконаних правил. Для цього кожен з двох наборів даних був перевірений на відповідність одній з трьох основних математичних моделей: лінійної (ф.1), гіперболічної (ф.2) та параболічної (ф.3).

Рисунок Г.6 – Сторінка тез конференції

Таблиця 1
Результати тестування

Кіл-ть правил	Drools, опр/с	EasyRules, опр/с	Кіл-ть правил	Drools, опр/с	EasyRules, опр/с
1	16818,185	99238,699	6	12951,352	18427,959
2	15503,489	53274,904	7	12365,963	15952,784
3	14385,258	38063,044	8	11840,686	14452,374
4	14918,214	27452,328	9	11593,190	12663,882
5	13751,477	22811,659	10	11233,706	11044,308

$$y = a + bx \ (1)$$
$$y = a + b/x \ (2)$$
$$y = a + bx + cx^2 \ (3)$$

де α– вплив незалежний від змінної,

b, c– вплив залежний від змінної,

x– незалежна змінна.

Моделювання проводилося за допомогою матричного середовища обчислень MATLAB та бібліотеки APMonitor Optimization Suite. Для двигуна висновків найкращим чином підійшла лінійна модель (ф.4). Двигун висновків EasyRules можна описати за допомогою гіперболічної моделі (ф.5).

$$y = 16865{,}20 - 605{,}36x \ (4)$$
$$y = 2606{,}41 + 98093{,}22/x \ (5)$$

Для оцінки якості отриманих моделей використовувався коефіцієнт кореляції між реальними даними (табл.1) і розрахунковими даними моделі. Отримані коефіцієнти кореляції для Drools і EasyRules склали 0,988 і 0,999 відповідно. Це свідчить про високу якість отриманих математичних моделей і можливість їх використання для проведення досліджень.

Щоб визначити який із програмних продуктів працює швидше, були досліджені їх математичні моделі. Було висунуто припущення, що існує деяка кількість послідовно виконуваних правил при якій кількість операцій в секунду буде однакова для обох моделей (ф.6).

$$16865{,}20 + 605{,}36x = 2606{,}41 + 98093{,}22/x \ (6)$$

Отримане рівняння було скорочено до форми алгебраїчного квадратного рівняння (ф.7).

$$605{,}36x^2 - 14258{,}79x + 98093{,}22 = 0 \ (7)$$

Значення дискримінанту квадратного рівняння (ф.7) менше нуля. Це означає, що дане рівняння не має рішення. Математичні моделі двигунів висновків не перетинаються. Діаграма математичних моделей (рис.2) наочно показує, що дві моделі не мають точки перетину.

Таким чином в результаті проведеного дослідження було доведено, що менш складна система керування бізнес-правилами EasyRules не поступається за сво-

Рисунок Г.7 – Сторінка тез конференції

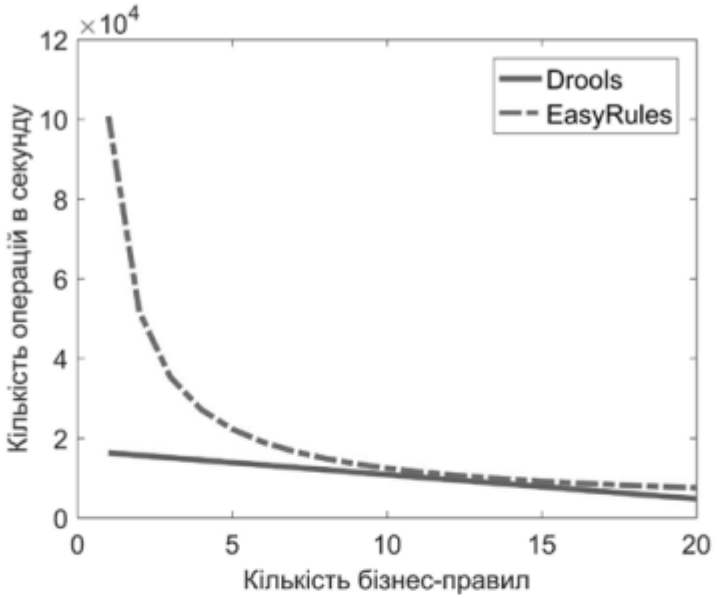


Рисунок 2 – Діаграма математичних моделей двигунів висновків

єю ефективністю складнішій системі Drools. Отримані математичні моделі доводять перевагу EasyRules щодо кількості послідовно виконуваних правил, особливо при їх невеликій кількості. Двигун висновків EasyRules може бути використаний у невеликих проектах без втрати швидкодії програмної системи.

Використані джерела:

1. Forsyth R. The Architecture of Expert Systems. // Expert Systems: Principles & Case Studies. London, 1984. P. 9-17.
2. Джарратано Д., Райли Г. Экспертные системы: принципы разработки и программирование. – 4-е изд. Москва.: Вильямс, 2007. – 1152с.
3. What is a Business rule management system?: [Електронний ресурс] // Medium – Режим доступу: https://medium.com/@ryanjollyyoung/what-is-a-business-rule-management-system-39ebcb7900c7 Дата звернення: 15.05.2019)
4. Rattanasawad T, Saikaew KR, Buranarach M, Supnithi T. A review and comparison of rule languages and rule-based inference engines for the Semantic Web. // 2013 International Computer Science and Engineering Conference (ICSEC). / Bangkok, 2011. doi:10.1109/icsec.2013.6694743
5. Fobel A., Subramanian N. Comparison of the performance of Drools and Jena rule-based systems for event processing on the semantic web. // Software Engineering Research Management and Applications (SERA) 2016 IEEE 14th International Conference / Towson, 2016. P. 24-30
6. World's fastest rules engine [Електронний ресурс] // JAVAWORLD – Режим доступу: https://www.javaworld.com/article/2078197/world-s-fastest-rules-engine.html (Дата звернення: 16.05.2019)

Рисунок Г.8 – Сторінка тез конференції