

УДК 519.682.2

Разработка автоматных программ на базе определения требований

Шелехов В.И. (Институт систем информатики СО РАН, Новосибирский государственный университет)

Технология автоматного программирования ориентирована на разработку простых, надежных и эффективных программ для класса реактивных систем. Автоматная программа реализует конечный автомат в виде гиперграфа управляющих состояний. В качестве языка спецификаций автоматных программ предлагается язык продукций, применяемый для описания сценариев использования (use case) – одного из видов функциональных требований. Технология представлена в виде свода золотых правил программирования, определяющих правильный баланс в интеграции автоматного, предикатного и объектно-ориентированного программирования. Технология иллюстрируется на наборе примеров.

Ключевые слова: понимание программ, автоматное программирование, определение требований.

1. Введение

Автоматная программа определяется в виде конечного автомата и состоит из нескольких *сегментов кода*. Вершина автомата – *управляющее состояние* программы. Ориентированная гипердуга автомата соответствует некоторому сегменту кода и связывает одну вершину с одной или несколькими другими вершинами [18]. Сегменты кода конструируются из фрагментов, являющихся предикатными программами [4].

Автоматные программы принадлежат классу программ-процессов (реактивных систем), более сложному по сравнению с классом программ-функций. Технология построения надежных и эффективных программ-функций разработана в рамках исследований по предикатному программированию [2, 4, 15, 16, 33]. Технология автоматного программирования [13, 17, 18] интегрирована с технологиями предикатного и объектно-ориентированного программирования. Гиперграфовая структура автоматной программы обеспечивает высокую гибкость, выразительность и эффективность программ.

Наряду с операторным языком автоматных программ [18] используется язык спецификаций требований. *Требования* — совокупность утверждений относительно свойств разрабатываемой программы. Одним из видов требований являются *функциональные требования*, определяющие поведение программы. Их наиболее популярной формой являются *сценарии использования (use case)* [27]. В нашем подходе сценарии использования представлены в виде правил на языке продукций [11], обычно применяемом для систем искусственного интеллекта. Это простой язык с высокой степенью декларативности. Спецификация на этом языке компактна и легко транслируется в автоматную программу, что позволяет использовать его как язык автоматного программирования.

Базис автоматного программирования определяется в разд.3. Дается определение автоматной программы. Описывается структура класса реактивных систем. Определяется язык требований, используемый в качестве языка автоматного программирования. Технология автоматного программирования (разд. 4) представлена в виде свода золотых правил программирования, определяющих правильный баланс в интеграции автоматного, предикатного и объектно-ориентированного программирования. Технология иллюстрируется на наборе примеров построения автоматных программ в разд. 5–10.

Работа выполнена при поддержке РФФИ, грант № 12-01-00686.

2. Обзор работ

В мировой практике технология определения требований разрабатывается и применяется в основном для больших интернетовских информационных систем и систем телекоммуникации, а также в системной инженерии (системотехнике). Технология спецификации требований отражена в стандарте [29]. В нашем подходе определение требований рассматривается для всего класса реактивных систем.

Инженерия требований имеет длительную почти сорокалетнюю историю, в т.ч. и в нашей стране, в основном на предприятиях аэрокосмического комплекса. К сожалению, исследования в этом направлении велись независимо и слабо интегрированы с мировым опытом. Это, в частности, обнаруживается и по используемой терминологии. В соответствии с тезисом «программирование без программистов» [8], именно специалисты по разработке космических аппаратов, знающие «физику» создаваемых аппаратов, должны разрабатывать управляющие программы космических аппаратов, а не программисты. При внимательном анализе обнаруживается, что здесь речь идет о той части программирования, которая относится к разработке требований. В зарубежных фирмах по разработке информационных

систем разработку требований осуществляют специалисты – *инженеры требований*; иногда они составляют половину персонала [32].

Языками спецификации требований являются: естественный язык, английский (80% случаев), формализованное подмножество естественного языка с использованием аппарата онтологий (15%) и формальный язык (5%) [32]. Популярной проблематикой является трансляция требований с естественного языка на формальные языки. Формальными языками требований являются: RSML [31], Statechart [37] SDL[34], UML, ALBERT [22] и др. Большинство из них являются графическими. В работе [28] выявлены существенные недостатки языка UML при его использовании для спецификации требований производственных систем. Формальными языками спецификации требований являются также общеизвестные универсальные языки спецификаций: VDM, Z, B и др. Семейство темпоральных языков спецификаций также используется для спецификации требований программных систем, в частности, систем реального времени [25]; наиболее популярным для спецификации требований является язык LTL, см. например [23]. Язык определения требований, предложенный в настоящей работе, существенно отличается от перечисленных языков. Наиболее близок к нему разработанный в целях тестирования язык спецификаций реактивных систем [35].

Продукционные системы искусственного интеллекта [30] реализуются набором правил вида $\langle \text{условие} \rangle \rightarrow \langle \text{действие} \rangle$. Правила именно такого вида используются в настоящей работе для определения требований. Ранее язык продукционных правил не применялся для определения требований программных систем. Исключением является работа [20], где язык продукций используется в целях разработки гибридных систем, однако без осознания того, что продукционные правила являются требованиями к разрабатываемой системе. Другой разновидностью продукционных правил являются охраняемые действия (guarded actions). Языками охраняемых действий являются: язык универсального внутреннего представления для нескольких разнообразных языков спецификаций в целях верификации, моделирования и синтеза [24], а также язык описания аппаратуры BlueSpec [12].

3. Базис предикатного программирования

3.1. Понятие автоматной программы

Автоматная программа определяется в виде конечного автомата и состоит из нескольких *сегментов кода*. Вершина автомата соответствует некоторому *управляющему состоянию*. Ориентированная гипердуга автомата соответствует некоторому сегменту кода и связывает

одну вершину с одной или несколькими другими вершинами. В качестве примера автоматной программы рассмотрим модуль операционной системы, реализующий следующий сценарий работы с пользователем, показанный на рис.1.



Рис. 1 – Схема работы ОС с пользователем: программа и ее автомат

В управляющем состоянии `login` операционная система запрашивает имя пользователя. Если полученное от пользователя имя существует в системе, она переходит в управляющее состояние `passw`, иначе возвращается в состояние `login`. В состоянии `passw` система запрашивает пароль. Если поданная пользователем строка соответствует правильному паролю, то пользователь допускается к работе и система переходит в состояние `session`. При завершении работы пользователя система переходит в состояние `login`.

Состояние автоматной программы определяется значениями набора переменных, модифицируемых в программе, за исключением локальных переменных. Взаимодействие с *внешним окружением* автоматной программы реализуется через прием и посылку *сообщений*, а также через *разделяемые переменные*, доступные в данной программе и других программах из окружения программы. Операторы *ввода* и *вывода* рассматриваются как упрощенная форма операторов посылки и приема сообщений.

3.2. Класс программ-процессов

Автоматное программирование ориентировано на класс программ-процессов. Его не следует применять для программ-функций, где предпочтительны традиционные средства, а также технология предикатного программирования [2, 4, 15, 16, 33].

Любая программа-процесс является *реактивной системой*, реализующей взаимодействие с внешним окружением программы и реагирующей на определенный набор событий (сообщений) в окружении программы. Определим структуру класса программ-процессов, т.е. класса реактивных систем.

В общем случае программа-процесс определяется в виде композиции нескольких автоматных программ, исполняемых параллельно и взаимодействующих между собой через

сообщения и разделяемые переменные. Каждая из параллельно исполняемых программ определяется независимым автоматом.

Подклассом реактивных систем являются *гибридные системы*, соединяющие дискретное и непрерывное поведение. Часть переменных состояния гибридной системы соответствует непрерывным параметрам (типа **real**), изменение которых реализуется независимо от программы гибридной системы по определенным законам, обычно формулируемым в виде дифференциальных уравнений. Важнейшими подклассами гибридных систем являются контроллеры систем управления и временные автоматы.

Система управления реализует взаимодействие с объектом управления для поддержания его функционирования в соответствии с поставленной целью. Системы управления используются в аэрокосмической отрасли, энергетике, медицине, массовом транспорте и др. отраслях. На каждом шаге вычислительного цикла *контроллер системы управления* получает входную информацию из окружения и обрабатывает ее. Результаты вычисления используются для передачи управляющего сигнала для воздействия на объект управления. Типовая структура контроллера системы управления определена в работе [13].

Временной автомат реализует функционирование процесса, используя показания времени. Пересчет времени проводится вне автоматной программы (временного автомата). Рассматриваются различные модели автоматов с дискретным и непрерывным временем [21]. Временной автомат является *системой реального времени*, если взаимодействие с окружением должно удовлетворять временным ограничениям, что характерно для встроенных систем. В системах с жестким реальным временем непредоставление результатов вычислений к определенному сроку является фатальной ошибкой. Большинство систем управления являются встроенными системами.

Автоматная программа является *детерминированной*, если из каждой вершины автомата (управляющего состояния) исходит не более одной гипердуги. Для *недетерминированного автомата* допускается несколько гипердуг (сегментов кода), исходящих из одного управляющего состояния. При исполнении программы из данного управляющего состояния недетерминировано выбирается один из сегментов кода. Недетерминированный автомат становится *вероятностным*, если для каждой гипердуги определена вероятность ее выбора. Отметим, что недетерминизм реализуется для параллельной композиции автоматных программ. Параллельная композиция может быть представлена эквивалентной интерливинговой разверткой, являющейся недетерминированной последовательной программой [19].

Автоматная программа может быть составлена из частей, принадлежащим разным подклассам реактивных систем. Например, возможно сочетание вероятностных и недетерминированных автоматов в рамках одной программы. Системы реального времени в большинстве случаев являются системами управления. В дополнении к этому автоматная программа может быть частью *распределенной системы*. Отметим также возможность интеграции с объектно-ориентированной технологией, когда состояние автоматной программы реализовано как объект класса.

3.3. Язык требований

В качестве языка спецификаций программ-процессов предлагается язык продукций [30], применяемый для описания сценариев использования (use case) [27] – одного из видов функциональных требований. Этот простой язык с высокой степенью декларативности, характерной для языков логического программирования. Он позволяет писать компактные и хорошо понимаемые спецификации программ-процессов. Спецификация автоматной программы в виде набора правил легко транслируется в эффективную автоматную программу, что позволяет использовать язык спецификации требований как язык автоматного программирования.

Требование определяет один из вариантов функционирования автоматной программы и имеет следующую структуру:

$$\langle \text{условие}_1 \rangle, \langle \text{условие}_2 \rangle, \dots, \langle \text{условие}_n \rangle \rightarrow \langle \text{действие}_1 \rangle, \dots, \langle \text{действие}_m \rangle;$$

Условиями являются: управляющие состояния, получаемые сообщения, логические выражения. *Действиями* являются: простые операторы, вызовы программ, посылаемые сообщения и итоговые управляющие состояния. Требование является спецификацией некоторого сегмента кода или его части. Семантика требования следующая: если в данный момент времени истинны все условия в левой части требования, то последовательно исполняется набор действий в правой части. Формальная семантика исполнения требований строится на базе темпоральной логики: условия определены для текущего управляющего состояния, а результаты действий – для следующего. Далее ограничимся детерминированными программами: для исполнения выбирается первое правило с истинными условиями.

Управляющее состояние в качестве $\langle \text{условия} \rangle$ означает, что исполнение программы находится в данном управляющем состоянии. Оно должно быть первым в списке условий, и может быть опущено лишь в случае, когда автоматная программа имеет единственное управляющее состояние. Управляющее состояние в качестве $\langle \text{действия} \rangle$ – это следующее

управляющее состояние, с которого продолжится исполнение программы после завершения исполнения данного требования. Оно должно быть последним в списке действий.

Неблокированный прием сообщения реализуется конструкцией `<имя сообщения>(<параметры>)`. Ее значение – **true**, если из окружения получено сообщение с указанным именем. Оператор **receive** `<имя сообщения>(<параметры>)` реализует прием сообщения с ожиданием появления сообщения из окружения. Отметим, что конструкция вида **M: if** (`<сообщение>`) `<оператор>` **else #M** эквивалентна:

receive `<сообщение>; <оператор>`.

Оператор **#M** есть оператор **goto M**. Оператор **send** `m(e)` посылает сообщение `m` с параметрами – значениями набора выражений `e`.

В качестве примера рассмотрим требования для модуля, реализующего сценарий работы ОС с пользователем на рис.1.

Содержательное описание представлено в разд. 3.1.

Окружение.

message `Get(string str);` // получение строки от пользователя

Управляющие состояния: `login, passw, session;`

Требования.

`login, Get(str) → User(str : #login : #passw);`
`passw, Get(str) → Password(str : #passw : #session);`
`session → UserSession(), login.`

Здесь `User` и `Password` – гиперфункции [18] с двумя ветвями без результирующих переменных.

Тип **time** используется для переменных и констант, значениями которых являются показания времени. Оператор **set** `t`, эквивалентный оператору **time** `t = 0`, реализует установку таймера, переменной `t`. Ее изменение производится непрерывно некоторым механизмом, не зависящим от автоматной программы. Оператор **delay** `T` реализует задержку исполнения программы на время `T`; по истечению этого времени программа продолжит работу со следующего оператора.

Язык предикатного программирования P [4] расширяется описаниями классов и конструкцией `<объект>.<имя элемента класса>` для доступа к элементу некоторого объекта. Описание класса определяет переменные, константы и методы как элементы класса. Описание метода представляется в виде определения предиката. При наличии единственного объекта класса доступ к элементу объекта возможен просто через `<имя элемента класса>`.

С управляющим состоянием может быть ассоциирован инвариант: **inv** <логическое выражение>. Логическое выражение должно быть истинным, когда исполняемая программа приходит в данное управляющее состояние. Инвариант не вычисляется при исполнении программы и должен быть истинным априори. Инвариант повышает понимание программы, его можно также использовать для верификации.

Для требований следующего вида:

$$s1, \langle \text{условие}_1 \rangle, \dots, \langle \text{условие}_n \rangle \rightarrow \langle \text{действие}_1 \rangle, \dots, \langle \text{действие}_m \rangle, s2$$

$$s1 \rightarrow \langle \text{действие}_1 \rangle, \dots, \langle \text{действие}_m \rangle, s2$$

где *s1* и *s2* – управляющие состояния, иногда будем использовать, соответственно, другие формы записи требований:

$$s1: \langle \text{условие}_1 \rangle, \dots, \langle \text{условие}_n \rangle \rightarrow \langle \text{действие}_1 \rangle, \dots, \langle \text{действие}_m \rangle \#s2$$

$$s1: \langle \text{действие}_1 \rangle, \dots, \langle \text{действие}_m \rangle \#s2$$

4. Методы автоматного программирования

Конструирование автоматной программы реализуется следующей последовательностью этапов. Сначала формулируется постановка задачи в форме *содержательного описания* с фиксацией набора требований. Формализация задачи начинается с описания элементов *внешнего окружения*. Специфицируются переменные *состояния* автоматной программы. Определяются связи между переменными. На следующем этапе фиксируются *управляющие состояния*. Некоторые из них снабжаются инвариантами. Построение автоматной программы реализуется в виде набора *требований*. Каждое из них обеспечивает адекватную реакцию на определенное сообщение или событие. Процесс построения программы сопровождается ее верификацией относительно содержательного описания.

Набор методов построения хороших (простых, надежных, эффективных и т.д.) программ представим в виде свода *золотых правил программирования*, определяющих правильный баланс в интеграции автоматного, предикатного и объектно-ориентированного программирования. Собирателем золотых правил в далеких 1960-ых годах был Г.И. Кожухин, один из разработчиков транслятора Альфа [1].

Правило №1 Геннадия Кожухина: *Ошибки в программе надо искать как грибы. Нашел одну, ищи рядом*. Сложность программы неравномерно распределена по программе. Имеются участки повышенной сложности; там вероятнее ошибиться. Один такой участок часто содержит более одной ошибки.

Автоматное программирование универсально. Любая программа-функция может быть запрограммирована в виде автоматной программы, которая, однако, будет значительно

сложнее аналогичной предикатной программы, построенной обычными средствами. Отсюда следует **правило №2**: *не следует использовать автоматное программирование для программ-функций.*

Поскольку сегменты кода автоматной программы конструируются из частей, соответствующих программам-функциям, необходима адекватная интеграция разных стилей программирования. Не следует смешивать разные стили. **Правило №3**: *части сегментов кода, соответствующие программам-функциям, должны быть представлены вызовами программ за исключением случаев, когда часть сегмента сводится к одному простому оператору.* Значительный по размеру фрагмент кода загромождает программу. Вынесение его из программы и оформление независимой подпрограммой улучшает понимание исходной программы. Однако запроцедурирование простых операторов в составе автоматной программы дает обратный эффект – избыточная структуризация введением дополнительного уровня иерархии усложняет программу.

Сложность автоматной программы экспоненциально зависит от числа переменных состояния программы, а также от числа и характера связей между переменными. Для упрощения программы применяется методы объектно-ориентированного программирования, позволяющие спрятать внутри классов часть переменных состояния и связей между ними. **Правило №4**: *замените состояние программы (или его часть) объектом класса, скрывающего часть переменных и связей между ними внутри класса.* Объектно-ориентированная декомпозиция существенно снижает сложность и размер автоматной программы. Однако не всегда. **Правило №5**: *не используйте классов, если нечего скрывать.* Автоматная программа проще не станет. **Правило №6**: *Не следует прятать внутри класса автомат программы, т.е. управляющие состояния и сегменты кода, оставляя в интерфейсе лишь объекты внешнего окружения.* Это худший способ реализации автоматной программы, выворачивающий ее наизнанку.

Модифицируем программу на рис.1 в стиле switch-технологии А. Шальто [14]:

```
type STATE = enum(login, passw, session);  
STATE state = login;  
while (true)  
switch (state) {  
  case login:   if (exists(get_string()))  
                 state = passw  
                 else state = login  
  case passw:  if (valid(get_string()))  
                 state = session  
                 else state = passw  
  case session: сессия_пользователя();  
                 state = login  
}
```

В модифицированной программе имеется лишь одно управляющее состояние, соответствующее началу тела цикла **while**, а единственным сегментом кода является тело цикла **while**. Таким образом, управляющие состояния `login`, `passw` и `session` становятся значением переменной состояния `state` в модифицированной программе. В работе [18, разд. 4, рис.2] детальным сравнением исходной и модифицированной программ показано, что исходная программа проще. **Правило №7:** *не следует переводить управляющие состояния в состояние автоматной программы.* Следствием является **правило №8:** *управляющее состояние, используемое явно или неявно в содержательном описании программы должно быть воспроизведено в автоматной программе.*

Инварианты автоматной программы, ассоциированные с управляющими состояниями, принципиально отличаются от инвариантов циклов и инвариантов классов императивной программы. В типичном случае, когда не требуется оптимизации автоматной программы, управляющее состояние не содержит инварианта; иначе говоря, инвариант тождественно истинен. Циклы в автоматной программе имеют другую природу по сравнению с циклами императивной программы. **Правило №9:** *не рекомендуется смешивать разные стили программирования, в частности, использовать циклы типа **while** для конструирования автоматной программы.*

В данной работе представлены типовые методы разработки автоматных программ. В случаях, когда оптимизация автоматной программы требует изменения ее гиперграфовой структуры, следует использовать метод трансформации требований [17].

5. Функционирование лампочки

Пример взят из руководства по системе верификации Uppal [36]. Описывается простой временной автомат для включения и выключения лампочки.

Содержательное описание. Имеется кнопка для включения и выключения лампочки. При нажатии на кнопку лампочка включается. Повторное нажатие на кнопку выключает лампочку. При двойном нажатии (быстрым нажатием на кнопку дважды) лампочка включается и загорается ярче. Нажатие считается *двойным*, если второе нажатие запаздывает по отношению к первому не более чем на 4 условные единицы измерения времени.

Окружение.

message `press`; // нажатие кнопки моделируется посылкой сообщения `press`

Управляющие состояния:

- **off** – лампочка выключена;
- **low** – лампочка включена и горит обычным светом;
- **bright** – лампочка включена и горит ярким светом.

Управляющее состояние **bright** реализуется после двойного нажатия кнопки, **low** – после одинарного.

Состояние.

time y; // хранит время в условных единицах

После установки в 0 значение переменной последовательно увеличивается, моделируя процесс изменения времени. Изменение y реализуется вне программы.

Представленные ниже требования являются непосредственной формализацией содержательного описания требований.

Требования.

off, **press** → **set y, low**
 low, **press, y<5** → **bright**
 low, **press** → **off**
 bright, **press** → **off**

Требования непосредственно переписываются в программу.

Программа.

```
process Лампочка {
  off:  if (press) { y=0 #low }
        else #off
  low:  if (press) { if (y<5) #bright else #off }
        else #low
  bright: if (press) #off
         else #bright
}
```

Поскольку конструкция **off: if (press) <X> else #off** эквивалентна

off: receive press; <X>, программа преобразуется к следующему виду:

```
process Лампочка {
  off:  receive press; y=0 #low
  low:  receive press; if (y<5) #bright else #off
  bright: receive press; #off
}
```

6. Электронные часы с будильником

На примере автоматной программы «Электронные часы с будильником» [11] дадим иллюстрацию определения требований и построения автоматной программы.

Содержательное описание. На корпусе часов имеется три кнопки:

- Н (Hours) – увеличивает на единицу число часов;
- М (Minutes) – увеличивает на единицу число минут;
- А (Alarm) – включает и выключает будильник.

Увеличение часов и минут происходит по модулю 24 и 60 соответственно. Если будильник выключен, то кнопка А включает его и переводит часы в режим, в котором кнопки Н и М устанавливают не текущее время, а время срабатывания будильника. Повторное нажатие кнопки А переводит часы в режим с включенным будильником, в котором кнопки Н и М будут менять время на часах. В режиме с включенным будильником, если текущее время совпадает со временем будильника, включается звонок, который отключается либо нажатием кнопки А, либо самопроизвольно через минуту. Нажатие кнопки А в режиме с включенным будильником переводит часы в нормальный режим без будильника.

На следующем шаге разработки программы из содержательного описания выделяется описание внешнего окружения программы. Действия с часами целесообразно представить в виде класса Часы.

Окружение.

message Н, М, А; // нажатие кнопок Н, М и А моделируется сообщениями

```
class Часы {
nat hours, minutes; // текущее время (часы, минуты)
nat alarm_hours, alarm_minutes; // время срабатывания будильника
inc_h(); {...} // увеличить время на один час
inc_m(); {...} // увеличить время на одну минуту
inc_alarm_h(); {...} // увеличить время будильника на час
inc_alarm_m(); {...} // увеличить время будильника на минуту
bell_on() {...} // включить звонок
bell_off() {...} // выключить звонок
bool bell_limit() {...}; // звонок звонит уже минуту
...
}
```

Использование класса Часы позволяет существенно разгрузить и тем самым упростить автоматную программу. В качестве состояния автоматной программы используется одна переменная **t** (объект класса Часы) вместо четырех переменных, которые были бы использованы в версии автоматной программы без применения объектно-ориентированной технологии. Работа часов реализуется независимым параллельным процессом. Реализация методов `inc_h`, `inc_m` и других должна гарантировать отсутствие конфликтов по доступу к переменным класса.

Управляющие состояния:

- **off** – режим работы часов без будильника;
- **set** – режим установки будильника;
- **on** – режим работы часов с включенным будильником.

Данные управляющие состояния явно обозначены в содержательном описании требований.

Представленные ниже функциональные требования являются непосредственной формализацией содержательного описания требований.

Требования.

```

off, H → inc_h
off, M → inc_m
off, A → set
set, H → inc_alarm_h
set, M → inc_alarm_m
set, A → bell_on, on
on, H → inc_h
on, M → inc_m
on, A → bell_off, off
on, bell_limit → bell_off, off

```

Состояние.

Часы t;

Программа. Программа очевидным образом строится по требованиям. Она отличается от представленной в работе [18].

```

process Работа_часов_с_будильником {
  Часы t = Часы();
  off: if (H) { t.inc_h() #off }
      else if (M) { t.inc_m() #off }
      else if (A) { #set }
      else #off
  set: if (H) { t.inc_alarm_h() #set }
      else if (M) { t.inc_alarm_m() #set }
      else if (A) { t.bell_on() #on }
      else #set
  on: if (H) { t.inc_h() #on }
      else if (M) { t.inc_m() #on }
      else if (A) { t.bell_off() #off }
      else if (t.bell_limit()) { t.bell_off() #off }
      else #on
}

```

7. Система управления банкоматом

Содержательное описание. *Банкомат* предназначен для автоматизированной выдачи и приёма наличных денежных средств с использованием платёжной *банковской карты*, ассоциированной со *счетом* в одном из *банков*. На магнитной полосе карты закодированы: номер карты, даты начала и окончания действия карты и др. информация. В банкомате есть устройство считывания банковских карт, устройство выдачи наличных, устройство приема наличных денег, устройство печати чеков, клавиатура и дисплей. Используя данные, закодированные на карте, банкомат через сервер, связанный с банкоматом, по системе межбанковской связи получает доступ к счету, ассоциированному с картой.

Клиент инициирует транзакцию, когда вставляет банковскую карту в устройство считывания банкомата. Если система опознает карту, то она проверяет, не истек ли срок действия, совпадает ли введенный клиентом ПИН-код с тем, что хранится в системе, не числится ли данная карта утерянной. Клиенту даются три попытки для ввода правильного ПИН-кода, после третьей ошибки карта блокируется.

Если ПИН-код введен правильно, система предлагает клиенту выбрать одну из трех операций: снятие денег, получение справки или пополнение счета, ассоциированного с картой. Прежде чем выдать наличные, система проверяет, что на указанном счете достаточно денег, не превышен суточный лимит и что в банкомате имеется требуемая сумма. Если транзакция одобрена, то выдается запрошенная сумма, печатается чек и карта возвращается клиенту. Для одобренной транзакции получения справки печатается чек. Клиент может в любой момент отменить транзакцию, при этом карта немедленно возвращается.

Окружение.

message card, cardTaken;

Сообщение `card` поступает в программу управления банкоматом, как только клиент вставит банковскую карту в устройство считывания банкомата. Сообщение `cardTaken` посылается программе управления банкоматом сразу после изъятия клиентом возвращенной банковской карты из устройства считывания банкомата.

Класс `Card_ops` определяет набор методов – операций с банковскими картами, применяемых в программе `cashMachine` управления банкоматом.

```

class Card_ops {
  validCard( : #yes : #no); // проверяется правильность банковской карты
  check_pin( : #yes: #no); // запрашивается ПИН-код и проверяется его правильность
  block_card(); // карта блокируется: дальнейшие операции с ней через банкомат невозможны
  process give_money();
  process put_money();
  print_info(); // печатает чек о состоянии счета, ассоциированного с банковской картой
  return_card(); // карта возвращается клиенту в устройстве считывания карт банкомата
  hideCard(); // карта, возвращенная клиенту, поглощается банкоматом
  ... // поля и методы скрытой части класса
}

```

Гиперфункция `validCard` проверяет, что вставленная карта является правильной банковской картой: карта с указанным на ней номером выдана в одном из банков, не просрочена, не заблокирована и не числится среди утерянных; выход `#yes` соответствует правильной карте, `#no` – если карта не прошла контроль. Процесс `give_money()` реализует выдачу денег клиенту. Сумма выдаваемых денег задается клиентом. Выдача денег реализуется при условии, что указанная сумма имеется на счете и в банкомате. В любой момент процесс может быть прерван клиентом. Процесс `put_money()` реализует прием денег от клиента через устройство приема наличных денег с зачислением их на счет, ассоциированный с картой. Метод `hideCard`, прячущий карту внутрь банкомата, срабатывает через время `Twait` после того, как карта была возвращена клиенту в устройстве считывания карт. Атрибуты банковской карты, считанные методом `validCard`, хранятся в скрытой части класса `Card_ops`.

Локальные программы.

```

hyper select( : #take : #put : #info : #cancel);

```

Гиперфункция `select` в соответствии с выбором клиента реализует переключение на одно из действий: `#take` – транзакцию выдачи денег, `#put` – транзакцию пополнения счета, ассоциированного с картой, `#info` – транзакцию получения справки, `#cancel` – завершение операций с банкоматом для получения банковской карты.

Управляющие состояния:

`idle` – банкомат находится в состоянии ожидания следующего клиента;

`transaction` – состояние банкомата перед выбором одного из трех видов транзакций по банковской карте, прошедшей авторизацию;

`take` – банкомат находится в транзакции выдачи денег;

`put` – банкомат находится в транзакции пополнения счета по банковской карте;

`info` – банкомат находится в транзакции выдачи справки о состоянии счета, ассоциированного с банковской картой;

cancel – банкомат находится в состоянии возврата клиенту его банковской карты .

Состояние. Объект класса Card_ops.

Требования.

```
process cashMachine {
  idle, card, → checkCard( : #transaction : #cancel);
  transaction → select( : #take : #put : #info : #cancel);
  take → give_money(), cancel;
  put → put_money(), cancel;
  info → print_info(), transaction;
  cancel → returnCard( : #idle)
}
```

Первоначально банкомат находится в состоянии **idle** ожидания очередного клиента. Банковская карта, вставленная в устройство считывания карт, через сообщение **card** инициирует запуск процесса **checkCard** (см. ниже), реализующего чтение атрибутов карты на магнитной полосе и проверку карты. Если карта правильная и верен введенный ПИН-код, реализуется переход в управляющее состояние **transaction**, иначе – в состояние **cancel**, где карта возвращается клиенту. В управляющем состоянии **transaction** клиент выбирает одну из трех транзакций или отказ от действий (кнопку Cancel) на основе чего гиперфункция **select** переключает работу банкомата на требуемую транзакцию или управляющее состояние **cancel**. В управляющем состоянии **take** запускается процесс **give_money** для выдачи клиенту денег со счета, ассоциированного с банковской картой. В управляющем состоянии **put** запускается процесс **put_money** для пополнения счета по банковской карте. В управляющем состоянии **cancel** запускается процесс **returnCard**, описанный ниже.

```
process checkCard( : #transaction : #cancel) {
  c0 → validCard( : #c1 : #cancel);
  c1 → check_pin( : #transaction: #c2);
  c2 → check_pin( : #transaction: #c3);
  c3 → check_pin( : #transaction: #c4);
  c4 → block_card(), cancel
}
```

Работа процесса **checkCard** начинается запуском гиперфункции **validCard** для считывания атрибутов банковской карты и проверки ее правильности. Если карта признана правильной, то вызывается гиперфункция **check_pin**, в которой клиенту предлагается ввести ПИН-код из четырех цифр; проверяется его правильность. Если ПИН-код верный, то процесс **checkCard** завершается выходом **#transaction**. В противном случае вызов **check_pin** реализуется еще два раза. После третьей ошибки карта блокируется вызовом метода **block_card**, после чего процесс **checkCard** завершается выходом **#cancel**.

```
process returnCard( : #idle) {  
    r0 → return_card(), set t, r1  
    r1, cardTaken → idle;  
    r1, t > Twait → hideCard(), idle;  
}
```

Процесс `returnCard` начинается вызовом метода `return_card`, реализующего возврат банковской карты в устройстве считывания карт банкомата. Устанавливается таймер `t`. По истечении времени `Twait` возвращенная карта прячется внутри банкомата, если только ранее клиент не изъял ее из устройства считывания карт, о чем сообщается сообщением `cardTaken`. В любом случае процесс `returnCard` завершается выходом `#idle`.

Программа процесса `cashMachine` строится по требованиям очевидным образом.

8. Программа управления лифтом

Современный пассажирский лифт [7] является сложным техническим сооружением. Управление работой лифта реализуется нетривиальной программой. Ее нередко используют для демонстрации технологии программирования. Примеры программ управления лифтом можно найти в работах [3, 5, 6, 12].

Содержательное описание. *Лифт* установлен в здании с несколькими *этажами*. Этажи пронумерованы. Лифт либо стоит на одном из этажей с *открытой* или *закрытой* дверью, либо находится между этажами и движется *вверх* или *вниз*.

На каждом этаже есть две *кнопки* вызова лифта: одна – для движения *вверх*, другая – для движения *вниз*. На нижнем этаже нет кнопки для движения *вниз*, а на верхнем – для движения *вверх*.

Внутри *кабины лифта* есть кнопки с номерами этажей. Нажатие одной из этих кнопок определяет команду остановки по прибытии лифта на соответствующий этаж.

Нажатые кнопки на этажах и в кабине лифта определяют текущее множество *заявок* на обслуживание пассажиров лифта. В момент завершения выполнения заявки соответствующая кнопка отжимается. Лифт может находиться в одном из трех состояний: направлении движения *вверх* (*up*), направлении движения *вниз* (*down*) и нейтральном (*neutral*). Лифт движется в одном из направлений (*up* или *down*) до тех пор, пока существуют заявки, реализуемые в этом направлении; когда заявки заканчиваются, направление движения лифта меняется на противоположное. При отсутствии заявок в обоих направлениях лифт останавливается на текущем этаже (состояние *neutral*).

По прибытии на этаж лифт либо останавливается на этаже, либо проходит мимо без остановки. Остановка реализуется при наличии заявки по данному этажу, т.е. при нажатой

кнопке в кабине лифта, либо при нажатой кнопке на этаже, но не в противоположном направлении движению лифта.

Решение об остановке на этаже принимается заранее вблизи этажа на определенном расстоянии по специальным датчикам. Если принято решение об остановке, включается торможение и лифт останавливается.

В случае остановки на этаже дверь лифта открывается. Закрытие двери лифта происходит через промежуток времени T_{door} , либо при нажатии кнопки «закрыть дверь» в кабине лифта. Если обнаружены помехи при закрытии дверей, они повторно открываются.

Окружение.

Класс Лифт определяет набор методов – примитивов, используемых в программе Lift управления лифтом.

```
class Лифт {
  decision1( : #idle : #start : #open);
  decision2( : #idle : #start );
  starting(); // лифт начинает движение из состояния покоя вверх или вниз
  stopping(); // вблизи этажа включается торможение для остановки на этаже
  check_floor( : #move : #stop ); // вблизи этажа решается, остановиться или проехать мимо
  bool near_floor(); // = true, когда движущийся лифт оказывается вблизи очередного этажа
  openDoor(); // реализуется открытие дверей лифта
  closeDoor(); // запускается процесс закрытия дверей лифта
  bool closeButton(); // = true при нажатии кнопки «закрыть дверь»
  bool closedDoor(); // = true, если закрытие дверей лифта завершено
  bool blockedDoor(); // = true, если закрытие дверей лифта остановлено человеком на этаже
  // поля и методы скрытой части класса:
  type DIR = enum (up, down, neutral); // тип состояния движения лифта
  DIR dir; // состояние движения лифта
  type FLOOR = first_floor .. last_floor; // тип номера этажа
  FLOOR floor; // номер этажа, мимо которого лифт проехал или на котором остановился
  ....
}
```

Для лифта, стоящего с закрытой дверью на некотором этаже, гиперфункция `decision1` в зависимости от состояния кнопок определяет один из трех вариантов дальнейших действий: `idle` – оставаться в состоянии покоя, `start` – начать движение, `open` – открыть дверь. После закрытия дверей лифта, остановившегося на некотором этаже, гиперфункция `decision2` в зависимости от состояния кнопок выбирает один из выходов: `idle` – оставаться в состоянии покоя, `start` – начать движение.

Программа Lift управления лифтом использует только первые 11 методов класса Лифт. Остальная часть класса скрыта. Однако переменные скрытой части класса могут быть использованы в инвариантах. Отметим, что в скрытой части находится большая часть окружения программы, в частности, весь механизм работы с кнопками и их индикацией.

Скрыто также много других возможных особенностей функционирования лифта, например, возможность блокировки закрывающейся двери лифта нажатием кнопки на этаже.

Состояние. Объект класса Лифт.

Управляющие состояния программы Lift:

```
idle: inv dir = neutral; // лифт стоит на некотором этаже, двери закрыты
start: inv dir ≠ neutral; // лифт начинает движение в направлении dir
open; // лифт подошел к некоторому этажу или стоит на этаже некоторое время
```

```
process Lift {
  idle → decision1( : #idle : #start : #open);
  start → Movement( : #open);
  open → atFloor( : #idle : #start)
}
```

В управляющем состоянии `idle` лифт стоит. В соответствии с гиперфункцией `decision1` произойдет переход снова в `idle`, пока не будет нажата кнопка на одном из этажей. Разумеется, кнопку может нажать и пассажир, проснувшийся в кабине лифта. Если нажата кнопка на том же этаже, на котором стоит лифт, происходит переход в управляющее состояние `open`. Гиперфункция `decision1` также формирует новое значение переменной `dir`. Вызовы `Movement` и `atFloor` соответствуют процессам, которые определены ниже.

Управляющие состояния программы Movement:

```
start: inv dir ≠ neutral; // лифт начинает движение в направлении dir
move: inv dir ≠ neutral; // лифт движется в направлении dir
stop: inv dir ≠ neutral; // лифт движется, находясь вблизи некоторого этажа,
// и начинает торможение, чтобы остановиться.
```

```
process Movement( : #open) {
  start → starting(), move;
  move, near_floor() → check_floor( : #move : #stop);
  stop → stopping(), open;
}
```

В процессе `Movement` метод `starting()` инициирует начало движения лифта с переходом в управляющее состояние `move`, в котором метод `near_floor()` проверяет приближение движущегося лифта к очередному этажу. Когда лифт находится вблизи этажа, запускается гиперфункция `check_floor`, определяющая, нужно ли останавливаться на этаже. Процесс переходит в управляющее состояние `stop`, если остановка нужна. Метод `stopping` запускает торможение лифта, и процесс `Movement` завершается внешним выходом `open`.

Управляющие состояния программы atFloor:

```
open; // лифт стоит на некотором этаже с закрытыми или полукрытыми дверями
opened; // двери лифта открыты
close; // двери лифта закрывается
```

Состояние программы atFloor:

```

time t;
process atFloor( : #idle : #start) {
  open → openDoor(), set t, opened;
  opened, closeButton() or t ≥ Tdoor → closeDoor(), close;
  close, closedDoor() → decision2( : #idle : #start);
  close, blockedDoor() → open;
}

```

Процесс atFloor начинает работу в управляющем состоянии **open**. Вызов метода openDoor реализует открытие дверей лифта, устанавливается таймер t, и происходит переход в управляющее состояние **opened**. При нажатии на кнопку «закрыть дверь» или через промежуток времени Tdoor метод closeDoor запускает процесс закрытия дверей с переходом в управляющее состояние **close**. В соответствии с третьим и четвертым правилами ожидается одно из двух событий: полное закрытие дверей при истинности closedDoor или блокировка дверей при истинности blockedDoor(). В случае блокировки дверей лифта процесс переходит в состояние **open**, в котором двери повторно открываются. В случае полного закрытия дверей гиперфункция decision2 в зависимости от состояния кнопок определяет вариант дальнейшего поведения лифта: оставаться ли в состоянии покоя или начать движение. В зависимости от выбранного варианта гиперфункция завершается по одному из выходов **idle** или **start**. Поскольку оба выхода – внешние для процесса atFloor, процесс atFloor завершает свою работу с возвратом в процесс Lift.

Программа. Сначала построим программы трех процессов по их требованиям.

```

process Lift {
  idle: decision1( : #idle : #start : #open);
  start: Movement( : #open)
  open: atFloor( : #idle : #start)
}
process Movement( : #open) {
  start: starting() #move;
  move: if (near_floor()) check_floor( : #move : #stop);
  stop: stopping() #open;
}
process atFloor( : #idle : #start) {
  time t;
  open: openDoor(); set t; #opened
  opened: if (closeButton() or t ≥ Tdoor) { closeDoor() #close }
  #opened
  close: if (closedDoor()) decision2( : #idle : #start);
  if (blockedDoor()) #open;
  #close
}

```

Подставляя тела программ Movement и atFloor, после упрощений получаем окончательную программу:

```

time t;
process Lift {
  idle: decision1( : #idle : #start : #open);
  start: starting();
  move: if (near_floor()) check_floor( : #move : #stop);
  stop: stopping();
  open: openDoor(); set t;
  opened: if (closeButton() or t ≥ Tdoor) { closeDoor() #close }
           #opened
  close: if (closedDoor()) decision2( : #idle : #start);
           if (blockedDoor()) #open;
           #close
}

```

Приведенная программа управления лифтом на порядок проще аналогичных программ в работах [3, 5, 6, 12]. Программа Lift компактна и использует 11 простых программ-функций. Программа универсальна, поскольку вся специфика внешнего окружения (кнопок и их индикации, датчиков вблизи этажа, а также закрытия и блокировки дверей) находится в скрытой части класса Лифт. Ряд особенностей работы лифта, описанные в содержательном описании, также оказалась в скрытой части.

С каждой кнопкой ассоциирована логическая переменная и независимый процесс, присваивающий этой переменной значение true при нажатии кнопки. Обнуление этой переменной, также как включение и выключение лампочек индикации, реализуется внутри класса Лифт при работе примитивов класса. Отметим, что было бы ошибочным нагружать процесс для нажатия кнопки любыми другими функциями. В принципе, возможен конфликт по доступу к переменной для кнопки при взятии значения переменной процессом Lift. При этом не произойдет нарушения работы лифта, и поэтому нет необходимости применять средства синхронизации.

9. Протокол чередования битов

Содержательное описание. Протокол чередования битов (Alternating Bit Protocol, ABP) реализует передачу данных через ненадёжные каналы связи. Процесс *передатчик* S вводит из внешнего окружения потенциально бесконечную последовательность блоков данных с помощью сообщения $in(d)$, где d – блок данных. Передатчик S пересылает очередной блок d сообщением $a(n, d)$, где n – номер блока, другому параллельно функционирующему процессу – *приемнику* R. Получив сообщение $a(n, d)$, приемник R выводит блок d во внешнее окружение с помощью сообщения $out(d)$.

Сообщения между процессами S и R передаются через *ненадёжные каналы* связи: возможны повторы, потери и искажения сообщений, но не меняется их порядок. Надёжная передача данных осуществляется при помощи передаваемого синхронизирующего бита и повторов сообщений. Здесь мы определим более простую версию, протокол n-ABP, в котором синхронизация реализуется с помощью номера блока, а затем покажем, как этот протокол трансформируется в классический протокол ABP.

Для обеспечения надёжности протокол n-ABP реализует следующие дополнительные действия. Приемник S содержит независимый счетчик m номеров блока. Получив очередной блок сообщением $a(n, d)$ приемник проверяет, что его номер блока n совпадает с m , и лишь в случае $n = m$ пересылает его сообщением $out(d)$. Получение любого сообщения $a(n, d)$ подтверждается посылкой ответного сообщения $b(n)$ приемнику S. Получив сообщение $b(j)$, приемник сверяет j с номером текущего посланного блока n . Условие $j = n$ гарантирует доставку очередного блока с номером n . В этом случае передатчик переходит к запросу следующего блока данных; в противном случае, через промежуток времени T_{wait} повторяет посылку сообщения $a(n, d)$.

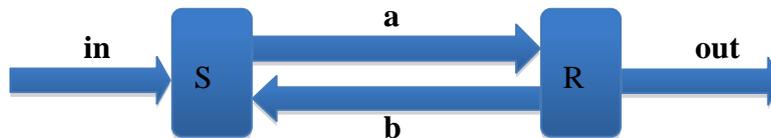


Рис.1. Схема протокола ABP

Моделирование ненадежной передачи сообщений a и b реализуется через логические переменные окружения sa и sb , значения которых непредсказуемо меняются во времени. Истинность переменной sa (или sb) определяет надежность передачи сообщения a (или b). Контроль порчи блока реализуется контрольным суммированием очередного блока и передачей этой суммы в сообщении a с последующей проверкой этой суммы в приемнике. Здесь мы опускаем действия по контролю порчи блоков, поскольку они не меняют принципиально схему протокола.

Окружение.

```

type Data;
message a(nat, Data), b(nat);
bool sa, sb;
  
```

Состояние: **nat** $n = 0, m = 0$; **time** t ; Data d

Формально следовало бы включить в состояние значение блока d в процессе R, однако фактически это локал.

Управляющие состояния: $s0, s1, s2, s3, s4, r0, r1, r3, r4$:

Требования.

process ABP() { S || R }

Оператор S || R реализует параллельное исполнение процессов S и R. При запуске протокола ABP счетчики n и m совпадают. Если один из процессов, S или R, будет остановлен, его перезапуск нельзя проводить автономно от другого процесса – это может привести к ошибке, поскольку счетчики n и m должны быть синхронизированы. Необходимо будет заново перезапускать протокол ABP (т.е. оба процесса S и R), либо использовать протокол рукопожатия для обеспечения синхронизации.

```

process S() {
  s0: nat n = 0 #s1
  s1: in(d) → #s2
  s2: set t #s3
  s3: sa → a(n, d) #s4
  s3: #s4
  s4: b(j), j = n → n = n+1 #s1
  s4: t>Twait → #s2
}
process R() {
  r0: nat m = 0 #r1
  r1: a(i, d) → #r3
  r3: i = m → out(d), m = m+1 #r4
  r3: #r4
  r4: sb → b(i) #r1
  r4: #r1
}

```

Программа.

```

process ABP() { S || R }
process S() {
  nat n = 0;
  s1: receive in(DATA d);
  s2: set t;
  if (sa) send a(n, d);
  s4: if (b(nat j) & j = n) { n = n+1 #s1}
  if (t>Twait) #s2 else #s4
}
process R() {
  nat m = 0;
  r1: receive a(nat i, DATA d);
  if (i = m) { send out(d); m = m+1};
  if (sb) send b(i);
  #r1
}

```

Анализируя программу протокола, можно определить, что номера блоков в сравнениях $j = n$ и $i = m$ отличаются не более, чем на единицу. Тогда переменные n и m можно заменить

логическими, $n = n+1$ заменить на $n = \neg n$, а $m = m+1$ – на $m = \neg m$. В результате получим классический протокол АВР.

10. Бытовой алгоритм: рыбная ловля

Приведенная ниже автоматная программа используется в работах [9, глава 4, стр. 47] и [10, глава 1, стр. 10] для иллюстрации основной алгоритмической структуры «силуэт» графического языка программирования Дракон. Программа проста и не требует предварительного содержательного описания. Интерес к этой программе определяется тем, что ее структура подобна соответствующей Дракон-схеме. Начнем с определения требований.

Требования.

process Рыбная_ловля(: #Конец) {

```

Подготовка_к_ловле:  Накопай_червей, Возьми_удочку,
                     Доберись_до_места_ловли #Начало_ловли
Начало_ловли:        Насади_червяка #Ожидание_клева
Ожидание_клева:      Забрось_удочку #Процесс_клева
Процесс_клева:       Клюнула → Подсекай #Рыбацкая_работа
Процесс_клева:       Пора_домой → #Обратная_дорога
Рыбацкая_работа:     Рыбка_попалась →
                     Сними_добычу_с_крючка_и_кинь_в_садок
                     #Продолжение_ловли
Рыбацкая_работа:     Пора_домой → #Обратная_дорога
Рыбацкая_работа:     Червяк_цел → #Ожидание_клева
Рыбацкая_работа:     #Начало_ловли
Продолжение_ловли:   Пора_домой → #Обратная_дорога
Продолжение_ловли:   #Начало_ловли
Обратная_дорога:     Собери_вещи #Дорога_домой
Дорога_домой:        Удалось_что-нибудь_поймать →
                     С_хорошим_настроением_направляйся_домой #Конец
Дорога_домой:        Зайди_в_магазин_купи_рыбы_чтобы_дома_не_краснеть
                     #Конец

```

}

Программа.

```

process Рыбная_ловля( : #Конец) {
    Накопай_червей;
    Возьми_удочку;
    Доберись_до_места_ловли;
    Начало_ловли: Насади_червяка;
    Ожидание_клева: Забрось_удочку;
    Процесс_клева: if (Клюнула) { Подсекай #Рыбацкая_работа }
                   else if (Пора_домой) #Обратная_дорога
                   else #Процесс_клева
    Рыбацкая_работа: if (Рыбка_попалась)
                     { Сними_добычу_с_крючка_и_кинь_в_садок
                       if (Пора_домой) #Обратная_дорога
                       else #Начало_ловли
                     }
                   else if (Пора_домой) #Обратная_дорога
                   else if (Червяк_цел) #Ожидание_клева
                   else #Начало_ловли
    Обратная_дорога: Собери_вещи;
                   if (Удалось_что-нибудь_поймать)
                       С_хорошим_настроением_направляйся_домой
                   else
                       Зайди_в_магазин_купи_рыбы_чтобы_дома_не_краснеть
                   #Конец
}

```

Данная программа изоморфна соответствующей Дракон-схеме, приведенной в работах [9, 10]. Каждому сегменту кода соответствует ветвь Дракон-схемы. Однако есть отличия. Для управляющих состояний **Начало_ловли** и **Процесс_клева** нет соответствующих ветвей в Дракон-схеме. Дополнительную ветвь **Начало_ловли** можно было бы внести в Дракон-схему, однако в этом случае она не поместилась бы на одной странице. Дополнительное управляющее состояние **Процесс_клева** введено из-за того, что внутри сегмента кода запрещены внутренние циклы, которые разрешены в Дракон-схеме.

Отметим, что требования и Дракон-схема не изоморфны. Набор управляющих состояний, используемых в требованиях, покрывает все ветви Дракон-схемы, но не наоборот.

Заключение

Автоматные программы по своей структуре существенно сложнее программ-функций. Технология автоматного программирования предлагает комплекс методов для упрощения автоматных программ в целях улучшения их понимания программистом.

Язык спецификации требований позволяет компактно формализовать логику процессов в виде набора правил. Каждое правило независимо определяет реакцию автоматной программы на внешнее событие.

Гиперграфовая структура автоматной программы, являющаяся продолжением аппарата гиперфункций, обладает более высокой степенью гибкости в декомпозиции программы, не реализуемой классическими автоматами. Гиперграфовая декомпозиция позволяет заменить информационные связи управляющими, упрощая тем самым состояние автоматной программы.

Автоматная программа строится из частей, относящихся к классу программ-функций. Технология автоматного программирования должна быть адекватно интегрирована с технологиями объектно-ориентированного и предикатного программирования. Набор рекомендаций по интеграции разных стилей программирования представлен в виде свода золотых правил программирования. Отметим, что описанная технология автоматного программирования может быть адаптирована для автоматного программирования на базе императивного языка вместо предикатного.

Для упрощения программы применяется методы объектно-ориентированного программирования, позволяющие спрятать внутри классов часть переменных состояния программы и связей между ними. Для программ, представленных в разд. 6–8, состояние автоматной программы полностью спрятано внутри соответствующего класса. Как следствие, в автоматной программе нет переменных и нет информационных связей между ее частями, что существенно ее упрощает.

В настоящей работе исследуется базис технологии автоматного программирования. Задачами следующего уровня являются:

- – разработка методов верификации автоматных программ;
- – специализация технологии автоматного программирования для разных подклассов реактивных систем;
- – разработка редактора для дуального (текстового и графического) представления автоматной программы; в качестве графического языка допустим только язык Дракон [9, 10].

Автор благодарен А.А. Шалыто за его работы по автоматному программированию, стимулировавшие исследования автора.

Работа выполнена при поддержке РФФИ, грант № 12-01-00686.

Список литературы

1. Бабецкий Г.И. и др. Система автоматизации программирования АЛЬФА // ЖВМиМФ, т.5, №2, М., 1965.

2. Вшивков В.А., Маркелова Т.В., Шелехов В.И. Об алгоритмах сортировки в методе частиц в ячейках // Научный вестник НГТУ, Т. 4 (33), 2008. С. 79-94.
3. Гома Х. UML. Проектирование систем реального времени, параллельных и распределенных приложений. М.: ДМК Пресс, 2002. 684 с.
4. Карнаухов Н.С., Першин Д.Ю., Шелехов В.И. Язык предикатного программирования Р. Новосибирск, 2010. 42с. (Препринт / ИСИ СО РАН; N 153).
5. Кнут Д.Э. Искусство Программирования. Том 1. Основные Алгоритмы. 2006. разд. 2.2.5.
6. Наумов А.С., Шалыто А.А. Система управления лифтом. Санкт-Петербург, 2003. 51с. [Электронный ресурс]. URL: http://is.ifmo.ru/download/elevator_a.pdf (дата обращения: 10.01.2015)
7. Манухин С.Б., Нелидов И.К. Устройство, техническое обслуживание и ремонт лифтов. М. «Академия», 2004. 336 с.
8. Паронджанов В. Д. Как улучшить работу ума. Алгоритмы без программистов – это очень просто! М.: Дело, 2001. 360 с.
9. Паронджанов В. Д. Учись писать, читать и понимать алгоритмы. Алгоритмы для правильного мышления. Основы алгоритмизации. М.: ДМК Пресс, 2012. 520 с.
10. Паронджанов В. Д. [Язык ДРАКОН. Краткое описание](http://drakon.su/media/biblioteka/drakondescription.pdf). М., 2009. 124 с. [Электронный ресурс]. URL: <http://drakon.su/media/biblioteka/drakondescription.pdf> (дата обращения: 10.01.2015)
11. Поликарпова Н.И., Шалыто А.А. Автоматное программирование / СПб.: Питер. 2009, 176с. [Электронный ресурс]. URL: <http://is.ifmo.ru/books/book.pdf> (дата обращения: 10.01.2015)
12. Решетников Е.О., Смачных М.В. Система управления пассажирским лифтом / Санкт-Петербургский государственный университет информационных технологий, механики и оптики. 2006. [Электронный ресурс]. URL: <http://is.ifmo.ru/download/umlift.pdf> (дата обращения: 10.01.2015)
13. Тумуров Э.Г., Шелехов В.И. Определение требований к системе управления полетом квадрокоптера // Тр. 16-й межд. конф. «Проблемы управления и моделирования в сложных системах». Самара, Самарский научный центр РАН, 2014. С. 627-633.
14. Шалыто А. А. SWITCH-технология. Алгоритмизация и программирование задач логического управления. СПб: Наука, 1998. [Электронный ресурс]. URL: <http://is.ifmo.ru/books/switch/1> (дата обращения: 10.01.2015)
15. Шелехов В.И. Верификация и синтез эффективных программ стандартных функций в технологии предикатного программирования // Программная инженерия, 2011, № 2. С. 14-21.
16. Шелехов В.И. Разработка и верификация алгоритмов пирамидальной сортировки в технологии предикатного программирования. Новосибирск, 2012. 30с. (Препринт / ИСИ СО РАН. № 164).
17. Шелехов В.И. Оптимизация автоматных программ методом трансформации требований // Тр. 2-й межд. конф. «Инструменты и методы анализа программ». Кострома, Костромской

- государственный технологический университет. 2014. С. 175-183. [Электронный ресурс]. URL:http://persons.iis.nsk.su/files/persons/pages/req_k.pdf (дата обращения: 10.01.2015)
18. Шелехов В.И. Язык и технология автоматного программирования // Программная инженерия, №4, 2014. С. 3-15. [Электронный ресурс]. URL: <http://persons.iis.nsk.su/files/persons/pages/automatProg.pdf> (дата обращения: 10.01.2015)
 19. Шелехов В.И. Тумуров Э.Г. Логика невзаимодействующих программ и реактивных систем // Вестник Бурятского Государственного Университета. Секция: математика, информатика, Вып. 9 / 2012. Улан-Удэ, 2012. С. 81-90.
 20. Шпаков В.М. Формализация и использование знаний о развитии процессов // Тр. 16-й межд. конф. «Проблемы управления и моделирования в сложных системах». Самара, Самарский научный центр РАН, 2014. С. 290-295.
 21. Alur R., Dill D.L. A theory of timed automata // Theor. Comput. Sci. 1994. P. 183-235.
 22. Aoumeur N., Saake G. Operational interpretation of requirements specification language ALBERT using timed rewriting logic // 5th International Workshop on Requirements Engineering: Foundation for Software Quality. 1999.
 23. Arcaini P., Gargantini A., Riccobene E. Online Testing of LTL Properties for Java Code // Hardware and Software: Verification and Testing, LNCS 8244. 2013. P. 95-111.
 24. Arvind H. Bluespec: a language for hardware design, simulation, synthesis and verification // MEMOCODE. 2003. P 249–254.
 25. Bellini P., Mattolini R., and Nesi P. Temporal logics for real-time system specification // ACM Comp. Surveys, 32(1). 2000. P.12–42.
 26. Brandt J., Gemünde M., Schneider K., Shukla S.K., Talpin J.-P. Representation of synchronous, asynchronous, and polychronous components by clocked guarded actions // Design Automation for Embedded Systems. 2012. P. 1 – 35.
 27. Cockburn A. Writing Effective Use Cases / Addison-Wesley. 2001. 270 P.
 28. Glinz M. Problems and Deficiencies of UML as a Requirements Specification Language // 10th International Workshop on Software Specification and Design. 2000. P. 11-22.
 29. IEEE Recommended Practice for Software Requirements Specifications. Revision: 29/Dec/11.
 30. Klahr D., Langley P., Neches R. Production System Models of Learning and Development. – Cambridge, Mass.: The MIT Press. 1987. 467 P.
 31. Leveson N., Heimdahl M., Hildreth H., Damon J. Requirements Specification for Process-Control Systems // IEEE Transactions on Software Engineering, 20 (4). 1994. P.684-707.
 32. Mich L, Franch M, Novi Inverardi P. Market research for requirements analysis using linguistic tools. Requirements Engineering 9(1). 2004. P. 40–56.
 33. Shelekhov V. I. 2011. Verification and Synthesis of Addition Programs under the Rules of Correctness of Statements // Automatic Control and Computer Sciences. Vol. 45, No. 7. P. 421–427.

34. Specification and description language (SDL). ITU-T Recommendation Z.100 (03/93). [Электронный ресурс]. URL: <http://www.itu.int/ITU-T/studygroups/com17/languages/Z100.pdf> (дата обращения: 10.01.2015)
35. Sunha A., Sharad M. Modeling Firmware as Service Functions and Its Application to Test Generation // Hardware and Software: Verification and Testing, LCNS 8244. 2013. P. 61-77.
36. A Tutorial on Uppaal 4.0. Revised and extended version. 2006. [Электронный ресурс]. URL: <http://www.uppaal.org/> (дата обращения: 10.01.2015)
37. Zhang W., Beaubouef T., and Ye H. "Statechart: A Visual Language for Software Requirement Specification // International Journal of Machine Learning and Computing. 2012. P. 52-61.

УДК 681.3:004.8

Сравнение системы «Discovery» с базовыми алгоритмами, встроенными в Microsoft SQL Server Analysis Services¹

Фирсов Н.И. (Институт систем информатики СО РАН)

В работе проводится сравнение системы «Discovery» с алгоритмами Microsoft Association Rules, Decision Trees и Neural Network, встроенными в Microsoft SQL Server Analysis Services. Показывается, что система «Discovery», во-первых, обладает теоретическими преимуществами перед этими алгоритмами, во-вторых, практически работает лучше на данных, где эти преимущества проявляются явно и, в-третьих, хорошо себя показывает на данных, взятых из репозитория UCI. Эти результаты демонстрируют определенные преимущества системы Discovery перед методами, встроенными в Microsoft SQL Server Analysis Services.

Ключевые слова: *Интеллектуальный анализ данных, извлечение знаний, предсказание, обнаружение закономерностей.*

1. Введение

В последнее время получили широкое развитие и активно применяются на практике различные KDD&DM-методы (Knowledge Discovery in Data Bases and Data Mining). Однако, используемые сейчас KDD&DM-методы имеют серьезные ограничения [1, 7]: каждый метод может работать только с определенными типами данных, имеет свой язык оперирования и интерпретации данных, и обнаруживает только определенный класс гипотез. Таким образом, они не способны извлекать из данных знания в полном объеме, а также могут получать результаты, не интерпретируемые в терминах предметной области.

Система «Discovery» реализует реляционный подход к методам извлечения знаний [1, 7, 11], снимающий упомянутые ограничения, свойственные KDD&DM-методам.

Система «Discovery» обладает следующими важными теоретическими свойствами: может обнаруживать теорию предметной области, может обнаруживать все правила, имеющие мак-

¹ Работа поддержана грантом РФФИ № 15-07-03410; интеграционными проектами СО РАН № 3, 87, 136, а также работа выполнена при финансовой поддержке Совета по грантам Президента РФ и государственной поддержке ведущих научных школ (проект НШ-3606.2010.1.)

симальные условные вероятности, может обнаруживать непротиворечивую вероятностную аппроксимацию теории предметной области [11], обнаруживает все максимально специфические правила, позволяющие предсказывать без противоречий [1, 12].

Наиболее близкими к системе «Discovery» методами можно считать поиск ассоциативных правил (Microsoft Association Rules) [10] и Decision Trees, в виду того, что закономерности в этих методах также представляются в форме логических правил. В данной работе ставится задача сравнения системы «Discovery» с Microsoft Association Rules, Decision Trees и Neural Network, встроенными в Microsoft SQL Server Analysis Services. Мы² покажем, что система «Discovery» обладает теоретическими преимуществами перед этими методами, практически работает лучше на данных, где эти преимущества явно проявляются и не хуже работает на реальных данных, взятых из DM-репозитория UC Irvine Machine Learning Repository (<http://archive.ics.uci.edu/ml/>).

Проведенное сравнение обосновывает необходимость реализации системы «Discovery» в виде плагина, подключаемого к службам Microsoft SQL Server 2005 Analysis Services (SSAS). Это позволяет использовать для сравнения алгоритмов единую среду разработки Business Intelligence Development Studio, единые средства визуализации Data Mining моделей, а также стандартные средства сравнения качества Data Mining моделей: диаграмму роста (Lift Chart) и классификационную матрицу (Classification Matrix).

2. Association Rules

Алгоритм Microsoft Association Rules состоит из двух шагов. Первый шаг – это ресурсоемкая фаза нахождения часто встречающихся наборов. Второй шаг – это генерация ассоциативных правил с использованием множества часто встречающихся наборов.

2.1. Нахождение часто встречающихся наборов

Под набором (itemset) мы понимаем набор истинностных значений предикатов. Например, $\{A(a) = 1; B(a) = 0; C(a) = 1\}$ – это набор длины 3. Запись a таблицы содержит некоторый набор, если на этой записи выполнены все предикаты данного набора. Поддержка набора (Support) – это количество записей таблицы, которые содержат данный набор.

Основным параметром, участвующим в нахождении часто встречающихся наборов, является параметр Minimum Support, который определяет, в каком минимальном количестве записей анализируемой таблицы должен содержаться некоторый набор, чтобы он являлся часто встречающимся.

На первой итерации находятся все часто встречающиеся наборы длиной 1. Алгоритм просто сканирует таблицу и подсчитывает поддержку каждого возможного предиката. Предика-

² Выражаю благодарность за помощь в работе и научное руководство, д.ф.-м.н. Витяеву Е.Е.

ты с поддержкой большей, чем *Minimum Support*, добавляются во множество часто встречающихся наборов длины 1. На второй итерации из часто встречающихся наборов, найденных на первой итерации, строятся всевозможные наборы длины 2, подсчитываются поддержки этих наборов, те наборы, которые проходят критерий *Minimum Support*, добавляются во множество часто встречающихся наборов длины 2. Далее из предикатов, входящих в часто встречающиеся наборы длины 2, строятся всевозможные наборы длины 3 и т.д. Алгоритм повторяется для наборов длины 3, 4, 5 и т.д., пока находятся наборы удовлетворяющие критерию *Minimum Support*.

Далее проверяется условие, что каждый поднабор часто встречающегося набора, также должен являться часто встречающимся набором.

2.2. Генерация ассоциативных правил

Следующая процедура генерирует ассоциативные правила:

1. Для любого часто встречающегося набора f , генерируем все поднаборы x и их дополнения $y = f - x$.
2. Если $Support(f) / Support(x) > Minimum\ Probability$, тогда $x \Rightarrow y$ является ассоциативным правилом с условной вероятностью $Prob = Support(f) / Support(x)$.

Параметр *Minimum Probability* задается перед началом обучения модели.

2.3. Прогнозирование

Следующий алгоритм по набору предикатов, поданных на вход, предсказывает значение целевого признака, либо выдает множество (n штук) наиболее вероятных значений целевого признака:

1. На вход подается некоторый набор предикатов. Ищутся все правила, условная часть которых совпадает либо с данным набором, либо с некоторым поднабором данного набора, а целевая часть содержит целевой признак. Найденные правила (k штук) применяются: целевые части правил и соответствующие условные вероятности добавляются в список рекомендаций.
2. Если подходящих правил не найдено, или их слишком мало ($k < n$), находятся $n - k$ наиболее популярных значений целевого признака. То есть, среди всех правил вида $\Rightarrow P = a_i$ (с пустой условной частью и целевым признаком в правой части) находятся $n - k$ правил с наибольшей условной вероятностью.
3. Предикаты, полученные на первых двух шагах, сортируются по вероятности.

3. Decision Trees

Основная идея алгоритма решающих деревьев состоит в рекурсивном разделении данных на подмножества, содержащие более или менее однородные состояния целевого (прогнозируемого) атрибута. При каждом разделении, все входные атрибуты оцениваются по их влиянию на целевой атрибут. Когда этот рекурсивный процесс заканчивается, решающее дерево сформировано.

3.1 Построение таблицы подсчета корреляций

Пусть для анализа с помощью алгоритма решающих деревьев дана некоторая таблица, с входными колонками F0, F1, F2, F3, и целевой колонкой P, размером 3000 записей, содержащую в своих ячейках только 0 или 1.

Тогда таблица подсчета корреляций на первом шаге алгоритма будет выглядеть следующим образом:

Таблица 1. Таблица подсчета корреляций.

		F0		F1		F2		F3	
		0	1	0	1	0	1	0	1
P	0	300	700	700	300	400	600	500	500
	1	200	1800	400	1600	400	1600	1100	900

Каждая колонка таблицы подсчета корреляций соответствует паре атрибут-значение одного из входных атрибутов. Каждая строка соответствует значению целевого атрибута.

Ячейки таблицы содержат количество корреляций соответствующих пар: входной атрибут-значение, целевой атрибут-значение. Например, пересечение первой строки и первого столбца таблицы подсчета корреляций содержит число 300, т.е. в анализируемой таблице есть 300 записей, у которых одновременно $F0 = 0$ и $P = 0$.

3.2 Нахождение наиболее подходящего для разбиения атрибута

Одним из широко известных критериев, с помощью которого можно найти необходимый атрибут, является *Энтропия*. Энтропия (H) определяется следующим образом:

$$H(p_1, p_2, \dots, p_n) = -p_1 \cdot \log(p_1) - p_2 \cdot \log(p_2) - \dots - p_n \cdot \log(p_n), \quad p_1 + p_2 + \dots + p_n = 1$$

Например, энтропия атрибута F1 равна: $H(F1) = H(700, 400) + H(300, 1600) = 0.946 + 0.629 = 1.571$

Атрибут с наименьшей энтропией является наиболее подходящим для разбиения. В данном примере первое разбиение будет по атрибуту F1. Решающее дерево после первого разбиения выглядит следующим образом:

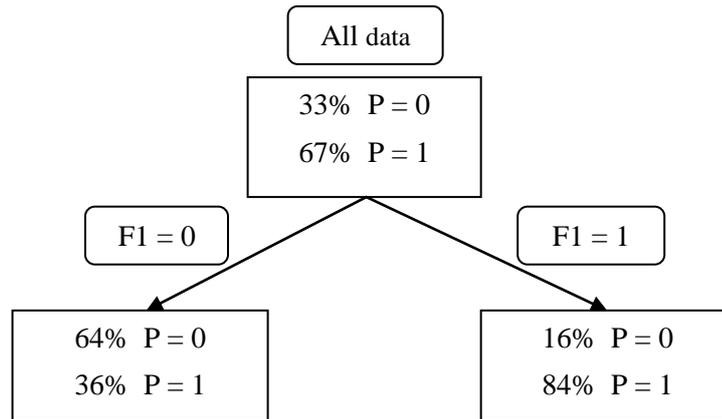


Рис. 1. Решающее дерево при разбиении по атрибуту F1.

Таким образом, разделив данные входной таблицы на два подмножества, далее рекурсивно применяем первый шаг алгоритма к новообразованным вершинам решающего дерева. Например, новая таблица подсчета корреляций для вершины решающего дерева $F1 = 0$ выглядит следующим образом:

Таблица 2. Таблица подсчета корреляций (после первого разбиения).

		F0		F1		F2		F3	
		0	1	0	1	0	1	0	1
P	0	300	700	700	0	400	600	500	500
	1	200	1800	400	0	400	1600	1100	900

3.3 Прогнозирование

Алгоритм предсказания решающих деревьев достаточно простой и эффективный. Поданный на вход предсказания набор предикатов, например $(F1 = 1, F2 = 0, F3 = 0, F4 = 1)$, спускается по дереву от корня к листьям по соответствующему этому набору пути, вершина дерева, находящаяся в конце этого пути и определяет предсказанное значение целевого признака. Таким образом, количество шагов в процессе прогнозирования не превышает максимальной длины пути от корня к листьям решающего дерева.

4. Neural Network

4.1. Структура нейронной сети

Нейронная сеть состоит из множества узлов (нейронов) и соединяющих их ребер. Есть три вида узлов: входные, скрытые и выходные узлы. Каждое ребро соединяет два узла и также имеет некоторый вес.

Входные узлы формируют первый уровень сети. Каждый входной узел сети связан с входным атрибутом. Значения входных атрибутов отображаются на отрезок $[-1, 1]$ и полученные числа подаются на соответствующие узлы первого уровня сети.

Скрытые узлы это узлы, составляющие промежуточные слои сети. Они получают на вход значения с первого (входного) слоя нейронной сети либо с предыдущего скрытого слоя. Каждый узел комбинирует значения, полученные с предыдущего слоя, с учетом весов соответствующих ребер, проводит некоторые вычисления и передает результирующее значение на следующий уровень сети.

Выходные узлы сети соответствуют выходным атрибутам модели. Результатом вычислений в таком узле обычно является число из отрезка $[0, 1]$, которое затем отображается на множество значений соответствующего выходного атрибута.

Заметим, что в случае, когда скрытые узлы отсутствуют, нейронная сеть представляет собой логистическую регрессию (logistic regression). То есть, скрытые узлы, это важные элементы сети, которые позволяют ей обнаруживать нелинейные закономерности.

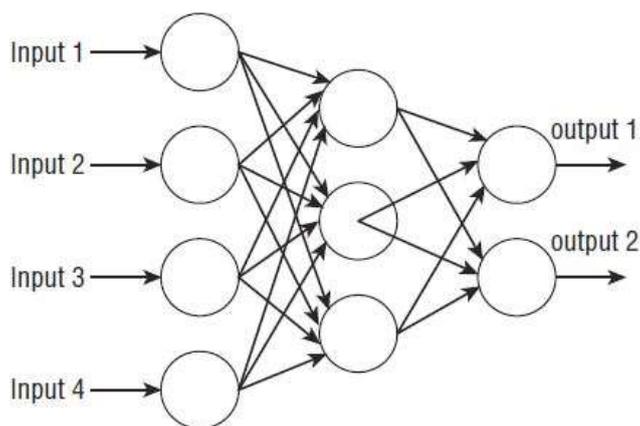


Рис. 2. Нейронная сеть со скрытыми слоями.

Каждый нейрон в нейронной сети является базовым вычислительным блоком. Нейрон имеет несколько входов и один выход. Он комбинирует все входные значения, производит вычисления и выдает на выход некоторое значение. Этот процесс похож на работу биологического нейрона.

Как показано на рисунке 3, нейрон использует две функции: одну для комбинирования входных значений и другую (так называемую функцию активации) для вычисления выходного значения. Существует несколько способов комбинировать входные значения, алгоритм Microsoft Neural Network использует для этого взвешенную сумму $w_1 \cdot \text{input}_1 + w_2 \cdot \text{input}_2 + w_3 \cdot \text{input}_3 + w_4 \cdot \text{input}_4$. Результат комбинирования входных значений подается затем в функцию активации. Алгоритмом Microsoft Neural Network используются две функции активации: \tanh в нейронах промежуточных (скрытых) слоев сети и sigmoid в нейронах выходного слоя.

Где $\tanh = (e^a - e^{-a}) / (e^a + e^{-a})$, $\text{sigmoid} = 1 / (1 + e^{-a})$. На рисунке 4 показаны графики этих функций.

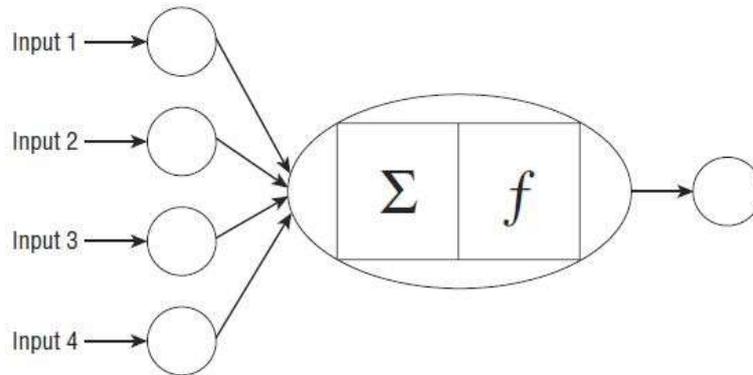


Рис. 3. Вычислительный блок нейронной сети (нейрон).

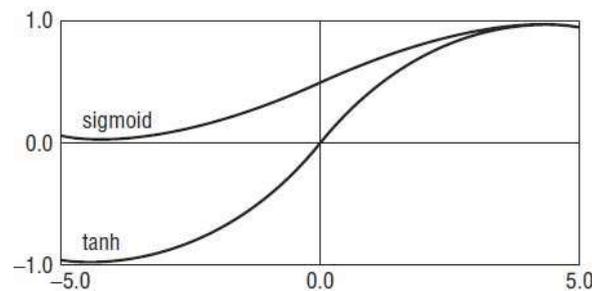


Рис. 4. График функций sigmoid и tanh.

4.2. Обучение.

Процесс обучения заключается в нахождении наилучшего набора весов для ребер сети, дающего минимальную ошибку предсказания.

1) Сначала все веса выставляются случайным образом (обычно в интервале от -1 до 1).

2) На каждом шаге обучения нейронная сеть, используя текущий набор весов, обрабатывает тренировочные записи, выполняя предсказание для каждой из них.

3) Далее, с помощью метода «обратного распространения ошибки» подсчитываются ошибки всех выходных и промежуточных (скрытых) нейронов, веса ребер сети корректируются.

4) Шаг 2 повторяется, пока не выполнится условие завершения обучения.

Ошибки нейронов выходного слоя считаются по формуле $Err_i = O_i(1 - O_i)(T_i - O_i)$, где O_i это выходное значение нейрона с номером i (т.е. предсказанное выходным нейроном значение), T_i это фактическое значение, которое нейрон должен был предсказать. Ошибки нейронов промежуточного (скрытого) слоя считаются по формуле $Err_i = O_i(1 - O_i) \sum_j Err_j w_{ij}$, где O_i это выходное значение нейрона с номером i , который имеет j ребер к

нейронам следующего слоя. Err_j – ошибка нейрона с номером j , w_{ij} – вес ребра между нейронами i и j .

После того, как ошибки всех нейронов были вычислены, веса ребер сети корректируются по формуле $w_{ij} = w_{ij} + l * Err_j * O_i$, где l величина от 0 до 1, называемая *скоростью обучения*. В процессе обучения величина l постепенно уменьшается для лучшей настройки весов сети.

Обучение останавливается, если достигнута достаточная точность на обучающем наборе, либо достигнут верхний лимит по количеству итераций обучения, либо веса после каждой итерации меняются меньше некоторой заданной границы.

4.3. Прогнозирование.

Хотя обучение нейронной сети это трудоемкий процесс, алгоритм предсказания достаточно эффективен. Поданные на вход значения атрибутов (например, Input1 = 2, Input2 = 4, Input3 = 5, Input4 = -1) нормализуются, и полученные значения записываются в нейроны входного уровня сети. Далее, нейроны скрытого слоя производят вычисления, как было описано выше, и полученные значения подаются на вход следующего скрытого слоя или выходного слоя. В конце, нейроны выходного слоя производят вычисления, и полученные числа отображаются (масштабируются) на множество значений соответствующих выходных атрибутов.

5. Система «Discovery»

Алгоритм поиска закономерностей системы «Discovery» реализует метод семантического вероятностного вывода, позволяющего находить все максимально специфические и максимально вероятные закономерности в данных [1]. Определим на высказываниях языка первого порядка вероятность, как описано в [6].

5.1 Семантический вероятностный вывод

Семантическим вероятностным выводом (СВВ) некоторого атома/литерала P является такая последовательность правил C_1, C_2, \dots, C_n , что:

$$1) C_i = (A_1^i \& \dots \& A_{k_i}^i \Rightarrow P), i = 1, \dots, n;$$

$$2) C_i \text{ является подправилом правила } C_{i+1}, \text{ т.е. } \{A_1^i, \dots, A_{k_i}^i\} \subset \{A_1^{i+1}, \dots, A_{k_{i+1}}^{i+1}\};$$

- 3) $\text{Prob}(C_i) < \text{Prob}(C_{i+1})$, $i = 1, 2, \dots, n-1$, где $\text{Prob}(C_i)$ – условная Вероятность (УВ) правила, $\text{Prob}(C_i) = \text{Prob}(P/A_1^i \& \dots \& A_{k_i}^i) = \text{Prob}(P \& A_1^i \& \dots \& A_{k_i}^i) / \text{Prob}(A_1^i \& \dots \& A_{k_i}^i)$;
- 4) C_i – Вероятностные Законы (ВЗ), т.е. для любого подправила $C' = (A_1 \& \dots \& A_j \Rightarrow P)$ правила C_i , $\{A_1, \dots, A_j\} \subset \{A_1^i, \dots, A_{k_i}^i\}$ выполнено неравенство $\text{Prob}(C') < \text{Prob}(C_i)$;
- 5) C_n – Сильнейший Вероятностный Закон (СВЗ), т.е. правило C_n не является подправилом никакого другого вероятностного закона.

Вероятностные неравенства в пунктах 3-4 проверяются на данных с помощью точного критерия независимости Фишера и критерия Юла [2, 3].

Множество всех цепочек СВВ предиката Р образуют дерево СВВ предиката Р.

Реализовать семантический вероятностный вывод в чистом виде не представляется возможным ввиду требований к производительности алгоритма, т.к. пункты 2 и 4 определения СВВ подразумевает большое пространство перебора. Для уменьшения перебора применяется следующие упрощения.

Во-первых, положим, что при построении цепочки СВВ правило C_{i+1} получается из правила C_i добавлением к его условной части только одного предиката. Эксперименты показывают, что крайне редка ситуация, когда добавление в условную часть правила сразу двух предикатов дает ВЗ, а добавление любого из этих двух признаков по отдельности не дает ВЗ. Следовательно, мы можем значительно уменьшить пространство перебора, почти не снижая количество и качество извлеченных из данных закономерностей.

Во-вторых, для того чтобы уменьшить перебор при проверке условия в пункте 4, используется поуровневая схема генерация правил: сначала генерируются все ВЗ с одним предикатом в условной части и заключением Р, затем с двумя предикатами, тремя и т.д. Таким образом, для проверки, является ли некоторое правило ВЗ, достаточно просмотреть все его подправила, находящиеся на предыдущем уровне дерева СВВ.

Перед началом обучения колонки входной таблицы помечаются атрибутами Input, PredictOnly и Predict, которые указывают, каким образом та или иная колонка участвует в обучении: в качестве входного признака, целевого признака, или в качестве обоих одновременно. Также в качестве параметров модели могут задаваться пороговые величины: условная частота правила, уровни значимости критериев Фишера и Юла, максимальное число интервалов значений для признака и др.

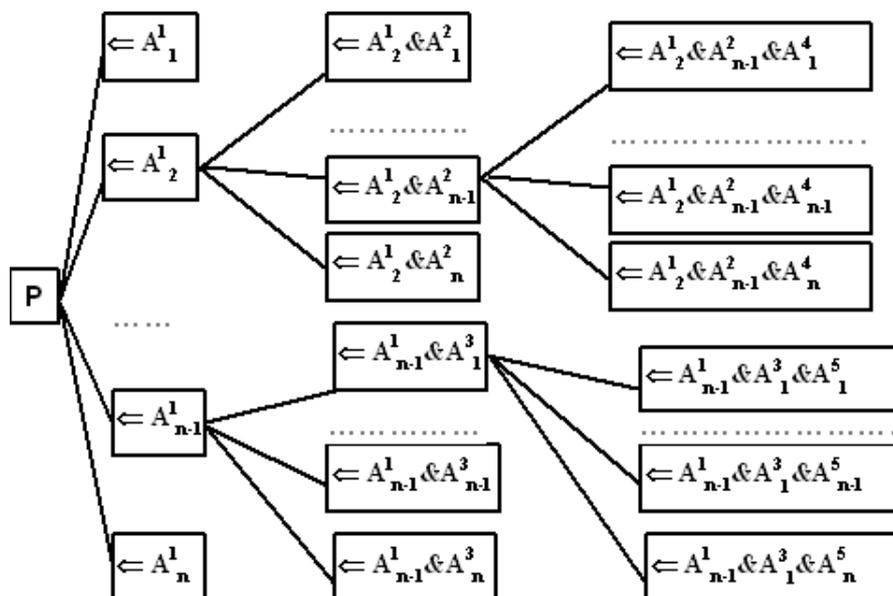


Рис. 5. дерево СВВ, включающее все СВВ, содержащие в заключении атом Р.

Результатом работы алгоритма является:

- 1) дерево СВВ для каждого целевого предиката;
- 2) множество ВЗ и СВЗ этих деревьев;
- 3) *Максимально Специфический Закон* (МСЗ) для каждого целевого предиката, определяемый как СВЗ, обладающий наибольшей условной вероятностью среди других СВЗ дерева вывода этого предиката.

Множество всех МСЗ обладает таким важным свойством как потенциальная непротиворечивость [1].

5.2 Алгоритм поиска закономерностей

1. Generate_First_Tree_Level (Queue Q, Tree T)

– Создаются все возможные правила, состоящие только из целевой части (они все по определению ВЗ). Для них сразу рассчитывается вся необходимая статистика на основе входных данных. Все элементы добавляются в корень дерева Т; элементы, содержащие Predict предикаты, добавляются в очередь Q.

2. Generate_Subsequent_Tree_Level (Queue Q, Tree T)

Обход дерева в ширину:

- Берем элемент из начала очереди Q, для него генерируем потомков, т.е. уточняем правило путем добавления 1-го нового предиката в условную часть.
- Проверяем, является ли новое правило ВЗ (см. Check_If_Probability_Low). Если является ВЗ, добавляем в дерево Т, добавляем в очередь Q.

– Процесс повторяется, пока очередь не пуста, т.е. еще можно получить новые ВЗ, путем добавления 1-го предиката в условную часть правила. Правила, соответствующие элементы которых не имеют потомков, являются СВЗ.

3. Check_If_Probability_Low (Rule R)

- Проверяем, является ли статистически значимым правило R с помощью критериев Фишера и Юла [1: с. 117-120]. Если нет, то R не является ВЗ;
- просматриваем все подправила Sr длины $\text{Length}(R) - 1$
- Если $\text{Prob}(Sr) > \text{Prob}(R)$, то R не является ВЗ;
- иначе R является ВЗ.

4. Extract_MSL (Tree T)

– Для каждого целевого предиката просматриваем его дерево СВВ. Сортируем множество СВЗ этого дерева по вероятности. Правила с наибольшей условной вероятностью являются Максимально Специфическими Законами (МСЗ) рассматриваемого целевого предиката.

5.3 Прогнозирование

Определим *меру правила* как неотрицательную величину, характеризующую качество и значимость правила в процессе предсказания.

Следующий алгоритм по набору предикатов, поданных на вход и множествам обнаруженных закономерностей ВЗ, СВВ и МСЗ, предсказывает значение целевого признака:

На вход подается некоторый набор предикатов. Ищутся все максимально специфичные закономерности, условная часть которых совпадает с данным набором либо с некоторым поднабором данного набора, а целевая часть содержит целевой признак. Максимально специфичная закономерность с наибольшей мерой определяет предсказанное значение целевого признака.

Если подходящих МСЗ не найдено, то рассматриваются все ВЗ с дерева СВВ целевого признака. Среди рассмотренных ВЗ ищутся те правила, условная часть которых совпадает с входным набором либо с некоторым поднабором входного набора. Правило с наибольшей мерой определяет предсказанное значение целевого признака.

В качестве меры правила можно использовать условную вероятность правила либо величину Юла. Величина Юла это число $U = Q - \text{NormalCDF}^{-1}(p) \cdot \sqrt{D}$, где

$$Q = \frac{f_{11}f_{22} - f_{12}f_{21}}{f_{11}f_{22} + f_{12}f_{21}}$$

есть коэффициент связи Юла, $D = \frac{1}{4} \cdot (1 - Q^2)^2 \cdot \left(\frac{1}{f_{11}} + \frac{1}{f_{12}} + \frac{1}{f_{21}} + \frac{1}{f_{22}} \right)$,

f_{ij} -частоты [3]. $\text{NormalCDF}^{-1}(p)$ - функция, обратная к функции нормального распределения. p - граничная вероятность, подается в качестве параметра алгоритма, используется также как граничная вероятность в критерии Фишера.

На многих тестах величина Юла в качестве меры правила дает лучшие результаты чем условная вероятность правила.

6. Data Mining Plug-in «Discovery»

Data Mining алгоритм "Discovery" реализован в виде плагина (Plug-in), подключаемого к службам Microsoft SQL Server Analysis Services (SSAS) посредством COM интерфейсов (см. рис.6). Это позволяет использовать Discovery наряду с другими алгоритмами, реализованными фирмой Microsoft, а также сравнивать качество Data Mining моделей, получаемых с помощью этих алгоритмов.

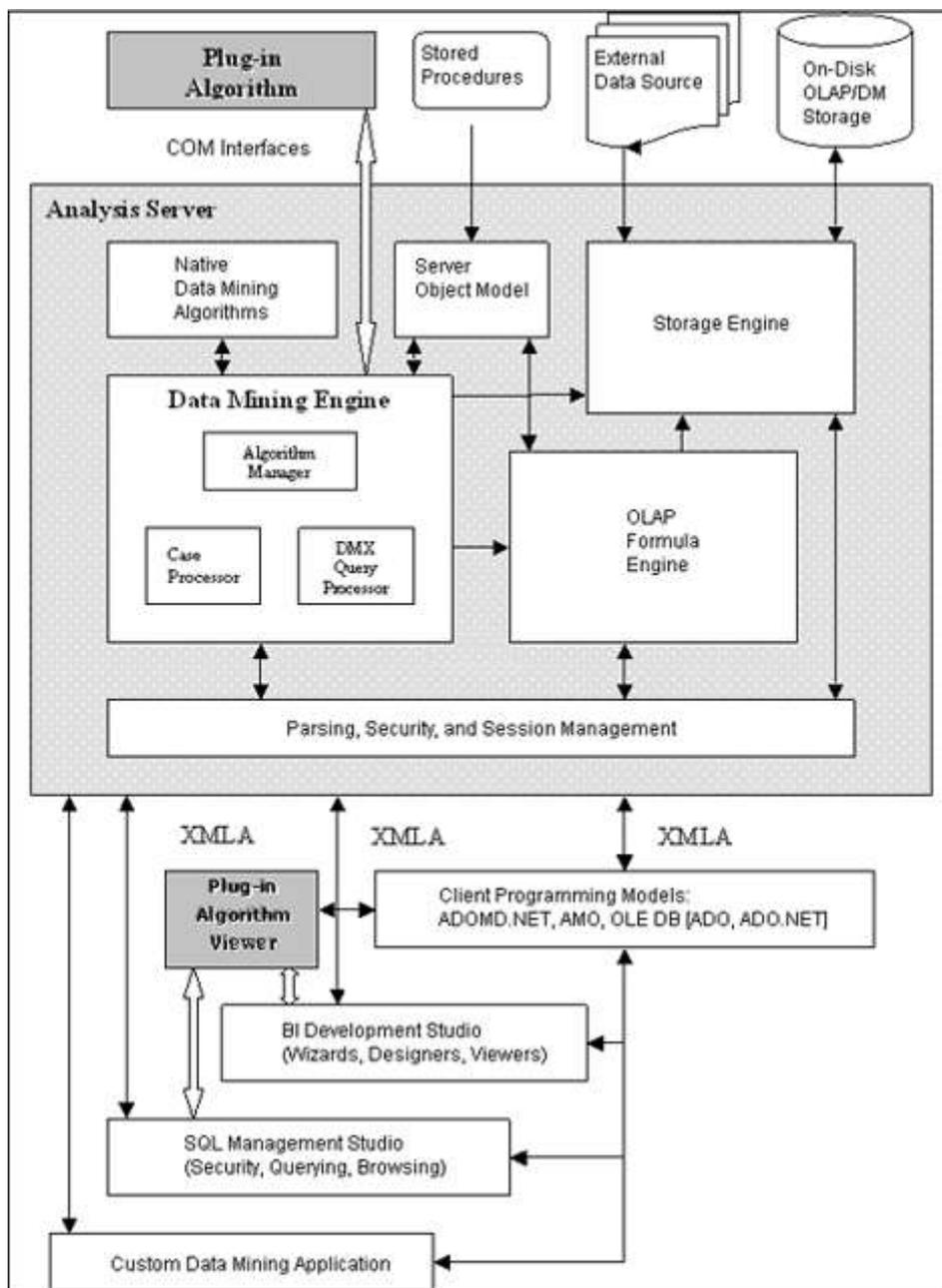


Рис.6. Архитектура Analysis Services.

Стандартным средством для работы с Data Mining плагинами SQL Server'a является среда разработки Business Intelligence Development Studio (BIDS), которая обладает инструментами для создания и обучения Data Mining моделей, визуализации и анализа качества моделей, предсказания.

Однако, возможностей BIDS для некоторых задач может быть недостаточно. Например, когда есть необходимость работать с Data Mining плагинами из стороннего приложения, для более глубокой автоматизации процесса обучения моделей и предсказания, для специфичной визуализации моделей и т.п.

Службы SQL Server Analysis Services поддерживают так называемую архитектуру с тонким клиентом. Среда разработки BIDS, среда SQL Management Studio, Excel и другие сторонние клиентские приложения обращаются к SSAS на языке XMLA (XML for Analysis).

XMLA-запросы могут содержать в себе DMX-запросы. Язык DMX (Data Mining Extensions to SQL) – SQL подобный язык запросов, позволяющий производить data mining-операции: создание и обучение модели, предсказание.

Функциональность XMLA базируется на двух основных функциях: Discover и Execute.

Метод Discover позволяет получать информацию о возможностях и состоянии провайдера данных (например DM-модели).

Метод Execute выполняет DMX запросы на сервере.

XMLA запрос использующий Execute выглядит следующим образом:

```
<Execute xmlns="urn:schemas-microsoft-com:xml-analysis">
  <Command>
    <Statement>
      CREATE MINING STRUCTURE [TestDMX]
      (
        [Key] DOUBLE KEY,
        [P] DOUBLE CONTINUOUS
      ) WITH HOLDOUT(30 PERCENT)
    </Statement>
  </Command>
  <Properties>
    <PropertyList>
      <Catalog>TestDS</Catalog>
      <Format>Tabular</Format>
      <Content>SchemaData</Content>
    </PropertyList>
  </Properties>
</Execute>
```

Клиентские DM-приложения на платформе .NET, работающие с Analysis Services, реализуются с помощью интерфейсов ADOMD.NET, которые инкапсулируют в себе создание и парсинг XMLA запросов. Среда SQL Server Management Studio позволяет выполнять XMLA и DMX запросы непосредственно на сервере.

6.1 Использование DMX для работы с Discovery Plugin

В данном разделе будет показано, как с помощью DMX и XMLA запросов можно создавать и обучать модель для алгоритма Discovery, а также производить прогнозирование.

1) Сначала необходимо подключиться к Analysis Server'у с помощью SQL Server Management Studio и создать новую базу данных (с именем, например, Discovery DMX Test).

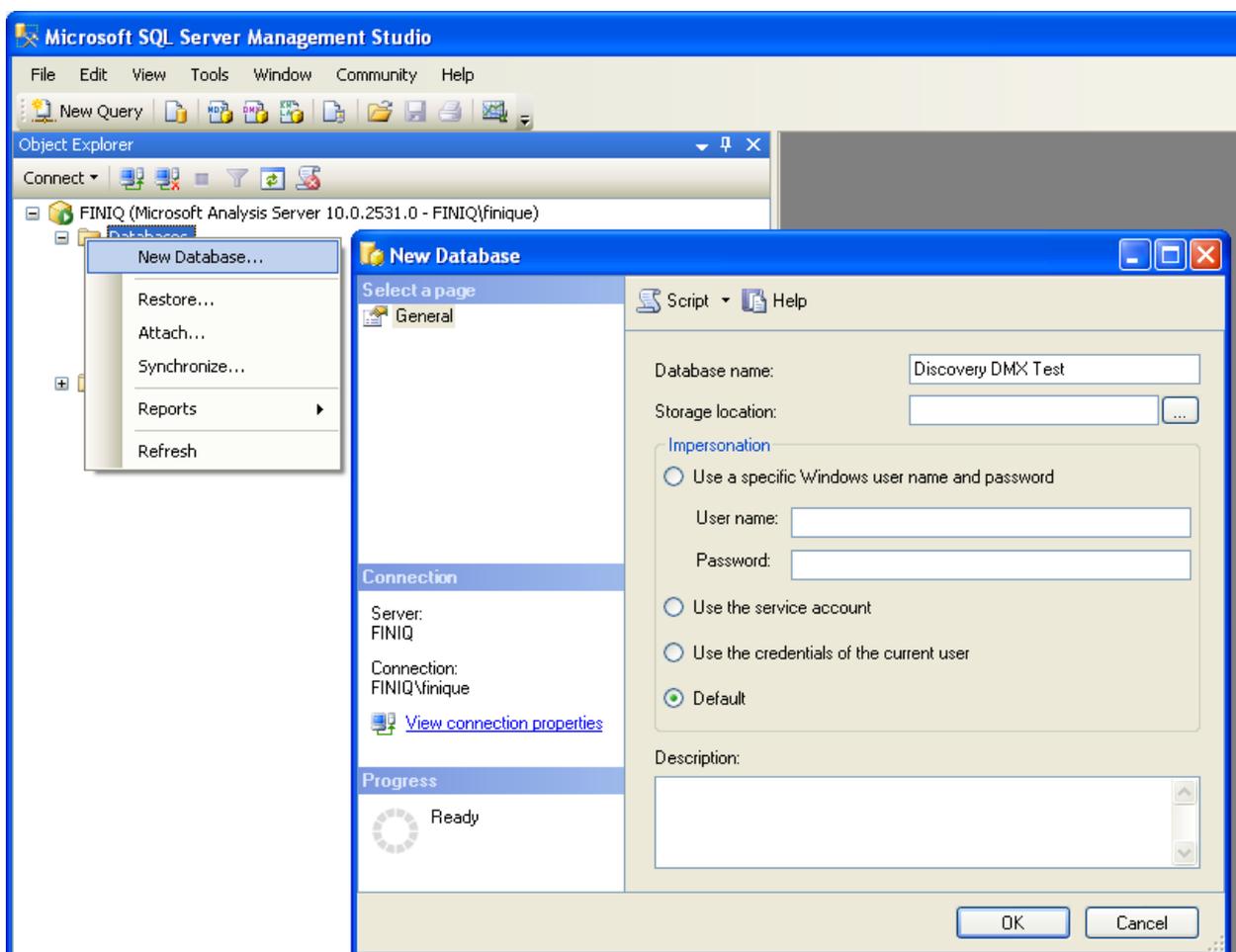


Рис. 7. Создание новой базы данных.

2) Затем с помощью DMX запроса создается Mining Structure, которая описывает задачу интеллектуального анализа данных (mining problem). Mining Structure определяет список используемых колонок, типы данных колонок и информацию о том, как с этими данными работать, т.е. являются ли они непрерывными, дискретными, циклическими и т.д.

```

CREATE MINING STRUCTURE [DiscoveryDMXTest]
(
  [Key] DOUBLE KEY,
  [F0] DOUBLE CONTINUOUS,
  [F1] DOUBLE CONTINUOUS,
  [F2] DOUBLE CONTINUOUS,
  [F3] DOUBLE CONTINUOUS,
  [F4] DOUBLE CONTINUOUS,
  [P] DOUBLE CONTINUOUS
)WITH HOLDOUT(0 PERCENT)

```

Выражение HOLDOUT(N PERCENT) делит данные на тренировочный и тестовый набор, для последнего случайным образом резервируется n процентов изначального набора данных.

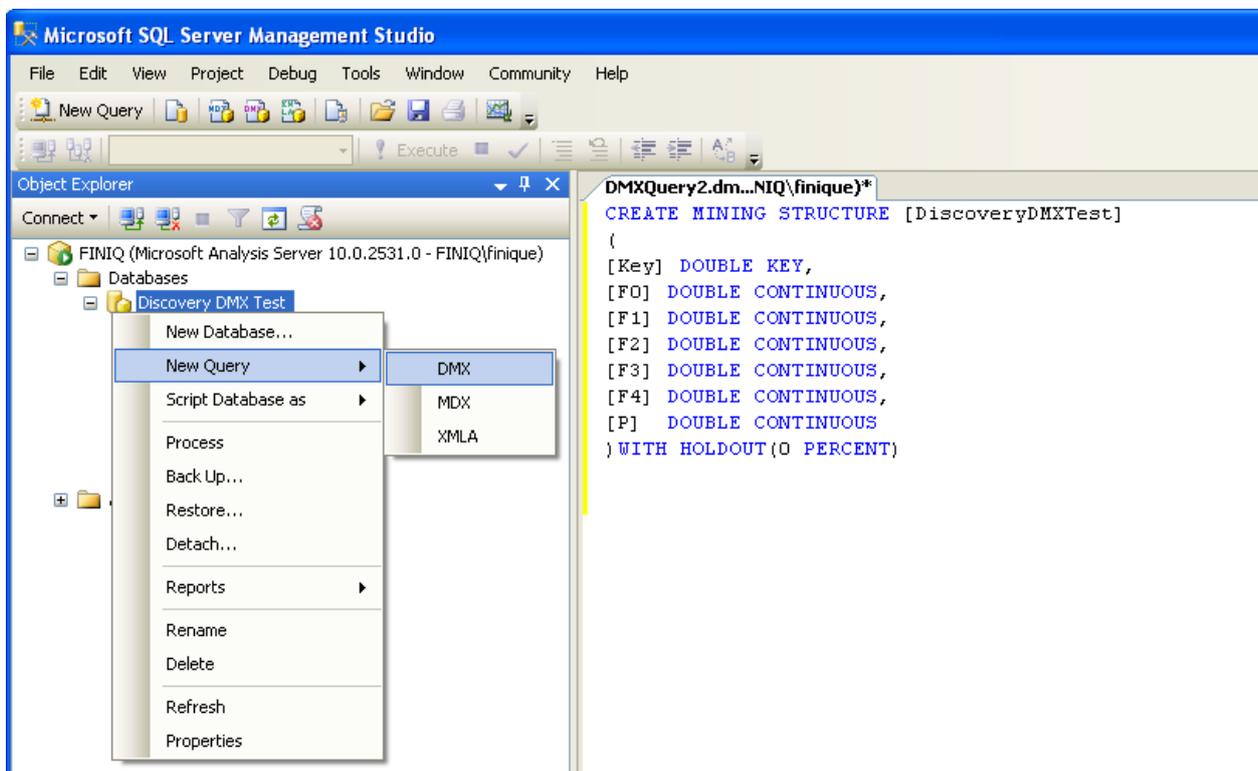


Рис. 8. Выполнение DMX запроса.

3) Следующий этап - создание Mining Model, которая определяет используемый DM-алгоритм, параметры алгоритма, а также как колонки используются в качестве DM-атрибутов (входной атрибут, предсказываемый атрибут, или оба значения одновременно).

```

ALTER MINING STRUCTURE [DiscoveryDMXTest]
ADD MINING MODEL [DiscoveryDMXTest]
(

```

```

[Key],
[F0],
[F1],
[F2],
[F3],
[F4],
[P] PREDICT_ONLY
)
USING Discovery_Plugin([Condition probability]=0.6)

```

2) Далее необходимо предоставить модели данные для тренировки. Для этого в базу данных (Discovery DMX Test) добавляется источник данных (DataSource), привязанный к файлу .mdb, из которого и будет заполняться DM-модель.

К сожалению, в DMX нет конструкций позволяющих создавать источники данных, но это можно сделать несколькими другими способами, мы воспользуемся XMLA:

```

<Create xmlns="http://schemas.microsoft.com/analysisservices/2003/engine">
  <ParentObject>
    <DatabaseID>Discovery DMX Test</DatabaseID> <!--Имя базы с п.2-->
  </ParentObject>
  <ObjectDefinition>
    <DataSource xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:ddl2="http://schemas.microsoft.com/analysisservices/2003/engine/2"
      xmlns:ddl2_2="http://schemas.microsoft.com/analysisservices/2003/engine/2/2"
      xmlns:ddl100_100="
        http://schemas.microsoft.com/analysisservices/2008/engine/100/100"
      xsi:type="RelationalDataSource">
      <ID>TestDS</ID> <!--ID Датасурса-->
      <Name>TestDS</Name> <!--Имя Датасурса-->
      <ConnectionString>Provider=Microsoft.Jet.OLEDB.4.0;DataSource=
        \TestDS.mdb</ConnectionString> <!--Путь к файлу mdb-->
      <ImpersonationInfo>
        <ImpersonationMode>Default</ImpersonationMode>
      </ImpersonationInfo>
      <Timeout>PT0S</Timeout>
    </DataSource>
  </ObjectDefinition>
</Create>

```

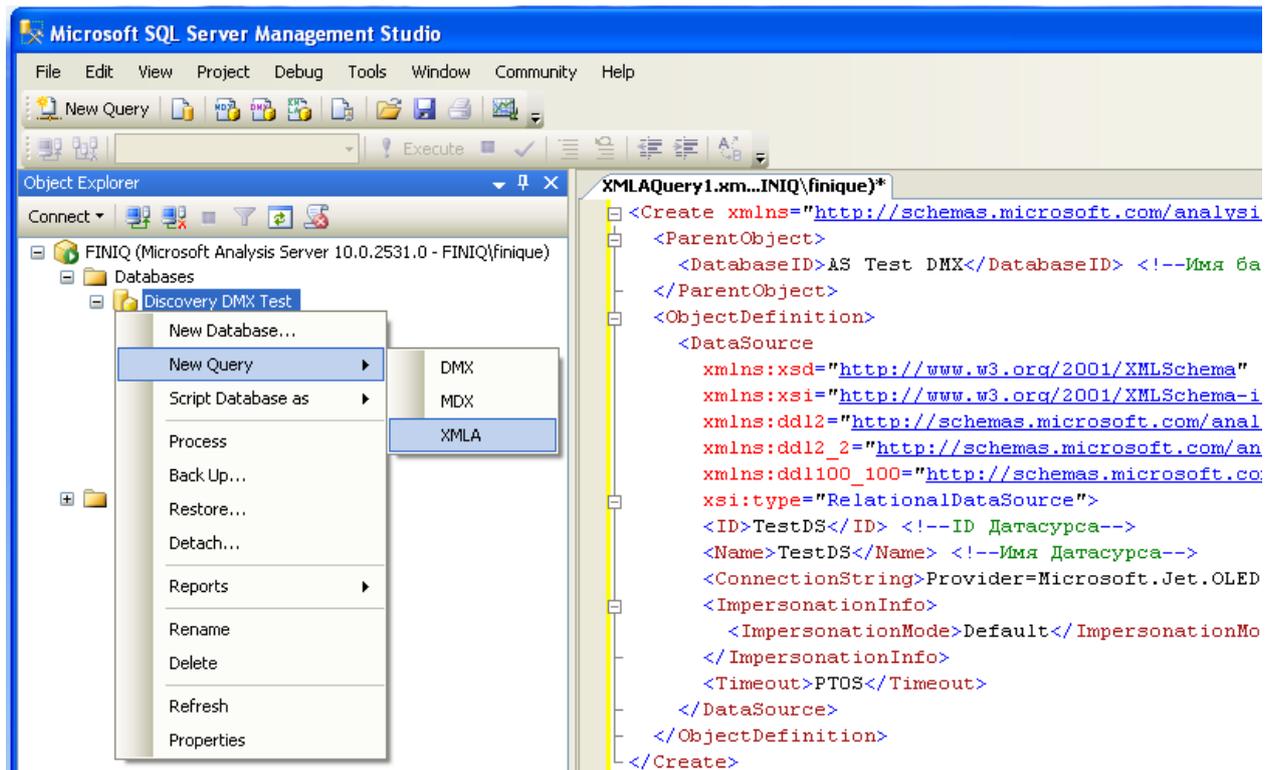


Рис. 9. Выполнение XMLA запроса.

Теперь заполняем модель из созданного источника данных:

```

INSERT INTO MINING STRUCTURE [DiscoveryDMXTest]
([Key], [F0], [F1], [F2], [F3], [F4], [P])
OPENQUERY (TestDS,
'SELECT [Key], [F0], [F1], [F2], [F3], [F4], [P]
FROM [TestDS_Table1]')

```

Данный DMX запрос одновременно производит обучение модели.

6.2 Прогнозирование

SQL Management Studio как и BIDS имеют удобный конструктор DMX запросов на предсказание, хотя, конечно, такие запросы можно писать вручную.

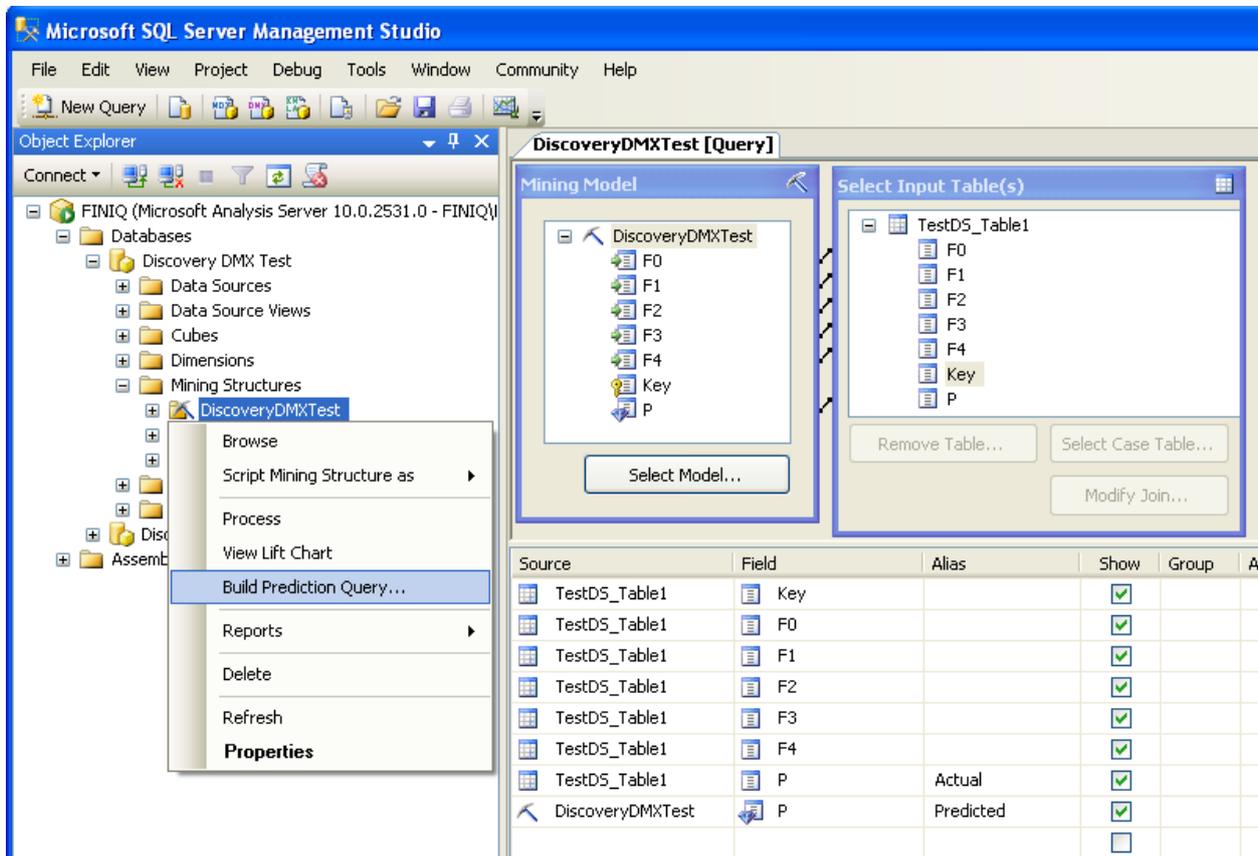


Рис. 10. Создание запроса на предсказание с помощью конструктора.

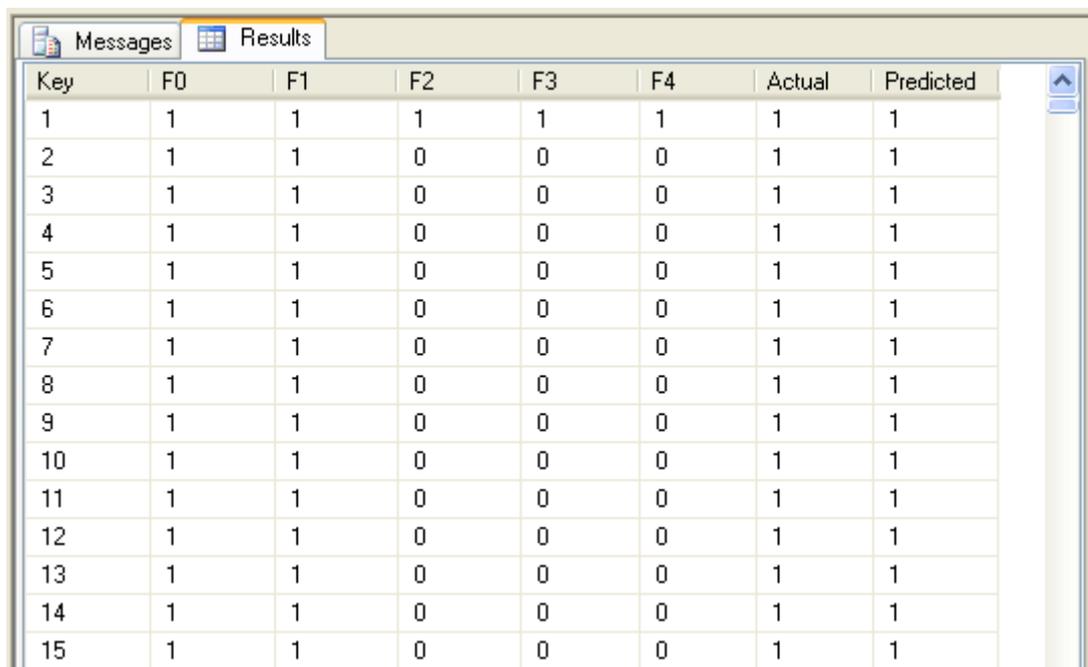
Следующий запрос выполняет функцию предсказания для всех записей таблицы TestDS_Table1 (на рис. 10. показано создание запроса). Запрос возвращает все колонки исходной таблицы, а также колонку предсказанных значений (Predicted). Далее можно сравнить действительные значения (колонка Actual) с предсказанными, подсчитать ошибку, сравнить с результатами других алгоритмов.

```

SELECT
    t.[Key], t.[F0], t.[F1], t.[F2], t.[F3], t.[F4],
    (t.[P]) as [Actual],
    ([DiscoveryDMXTest].[P]) as [Predicted]
From
    [DiscoveryDMXTest]
PREDICTION JOIN
    OPENQUERY ([TestDS],

```

```
'SELECT
  [Key], [F0], [F1], [F2], [F3], [F4], [P]
FROM
  [TestDS_Table1]
') AS t
ON
[DiscoveryDMXTest].[F0] = t.[F0] AND
[DiscoveryDMXTest].[F1] = t.[F1] AND
[DiscoveryDMXTest].[F2] = t.[F2] AND
[DiscoveryDMXTest].[F3] = t.[F3] AND
[DiscoveryDMXTest].[F4] = t.[F4] AND
[DiscoveryDMXTest].[P] = t.[P]
```



Key	F0	F1	F2	F3	F4	Actual	Predicted
1	1	1	1	1	1	1	1
2	1	1	0	0	0	1	1
3	1	1	0	0	0	1	1
4	1	1	0	0	0	1	1
5	1	1	0	0	0	1	1
6	1	1	0	0	0	1	1
7	1	1	0	0	0	1	1
8	1	1	0	0	0	1	1
9	1	1	0	0	0	1	1
10	1	1	0	0	0	1	1
11	1	1	0	0	0	1	1
12	1	1	0	0	0	1	1
13	1	1	0	0	0	1	1
14	1	1	0	0	0	1	1
15	1	1	0	0	0	1	1

Рис. 11. Результат выполнения запроса предсказания.

6.3 Плагин Discovery для Excel

Как уже было упомянуто ранее, архитектура Analysis Services позволяет работать с DM-плагинами (в том числе и Discovery) с помощью Microsoft Excel [5]. Excel дает возможность использовать всю описанную выше функциональность с помощью хорошо знакомого многим пользовательского интерфейса.

Excel позволяет создавать, обучать, просматривать и управлять DM-моделями. В качестве данных можно использовать таблицу Excel, либо некоторый выбранный диапазон, либо внешний источник данных. На рисунке ниже показано создание DM-модели для алгоритма Discovery в Excel.

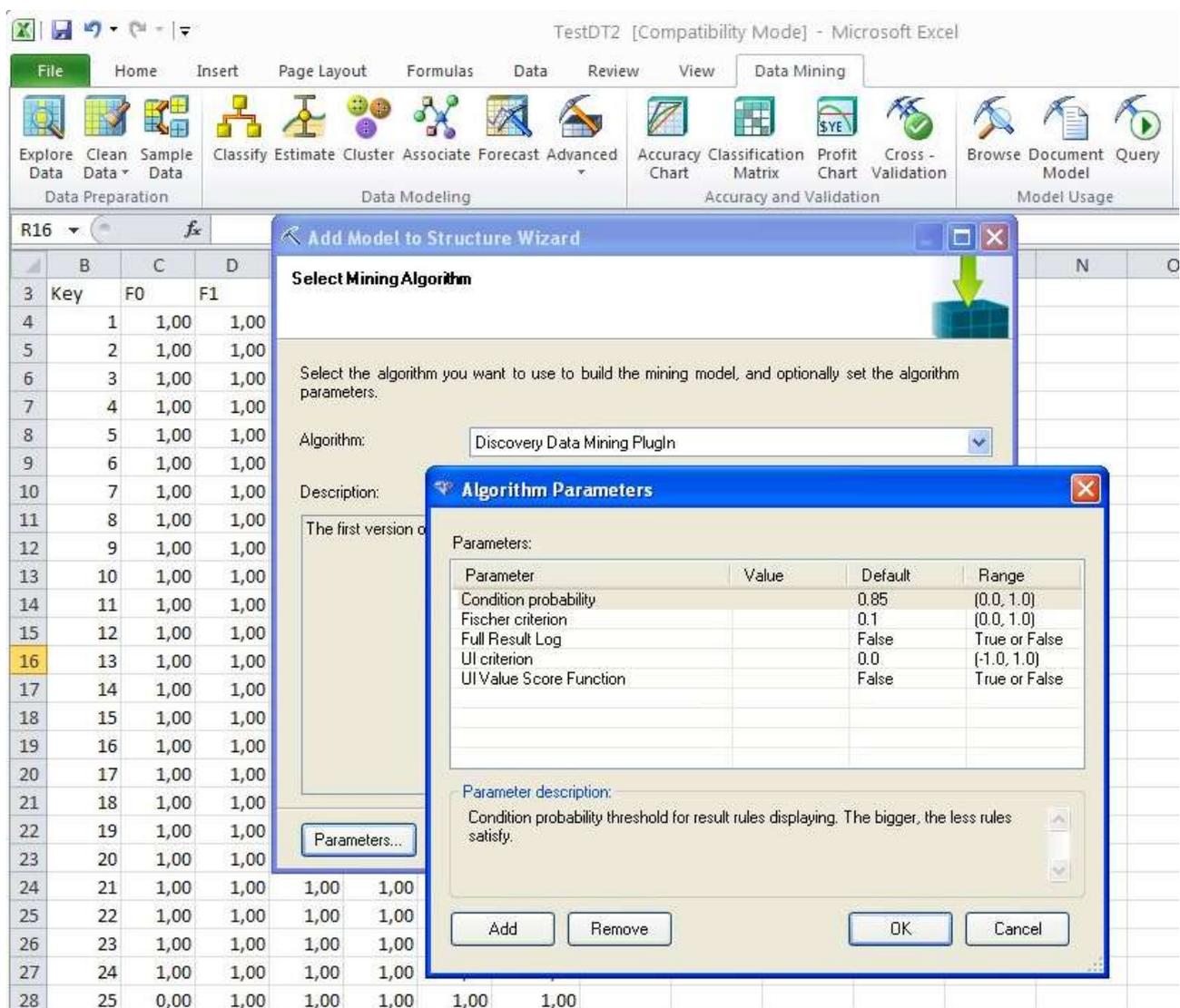


Рис. 12. Создание DM-модели для алгоритма Discovery в Excel.

Плагин «Discovery» имеет следующие параметры:

Condition probability – граничное значение условной вероятности. Правила с УВ меньшей данной границы не отображаются в списке правил, полученных в результате обучения модели.

Fischer criterion - граничная вероятность в критерии Фишера.

UI criterion – граничное значение величины Юла.

UI Value Score Function – определяет, использовать в качестве меры правила величину Юла или условная вероятность правила.

Для просмотра DM-моделей в Excel используется стандартное средство визуализации. Оно обладает очень ограниченной функциональностью, однако, есть возможность реализовать для плагина Discovery собственное средство просмотра (Plug-in Viewer), с необходимым набором функций. На рисунке ниже показано стандартное окно просмотра моделей с правилами, найденными в результате обучения модели с помощью алгоритма Discovery.

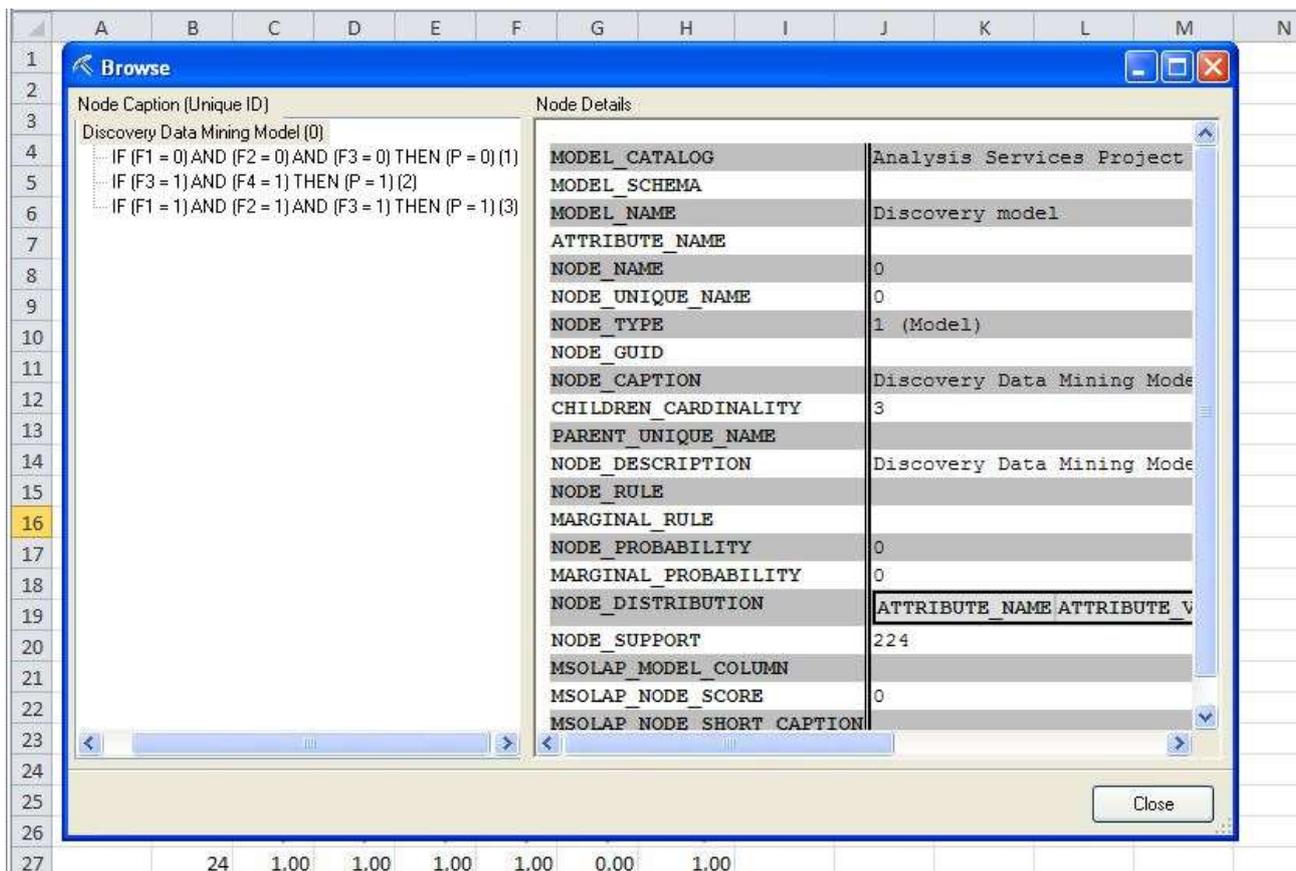


Рис. 13. Стандартное окно просмотра DM-моделей в Excel.

Excel позволяет оценивать точность предсказания DM-моделей с помощью таких известных инструментов, как Accuracy Chart, Classification Matrix, Profit Chart, Cross-Validation.

Также Excel обладает мощным средством для создания DMX запросов. С помощью мастера запросов можно создавать не только запросы на предсказание, но и множество других запросов для построения и управления DM-моделями, для чего в мастере содержится ряд дополнительных шаблонов. На следующем рисунке показан процесс создания запроса на предсказание с помощью мастера запросов.

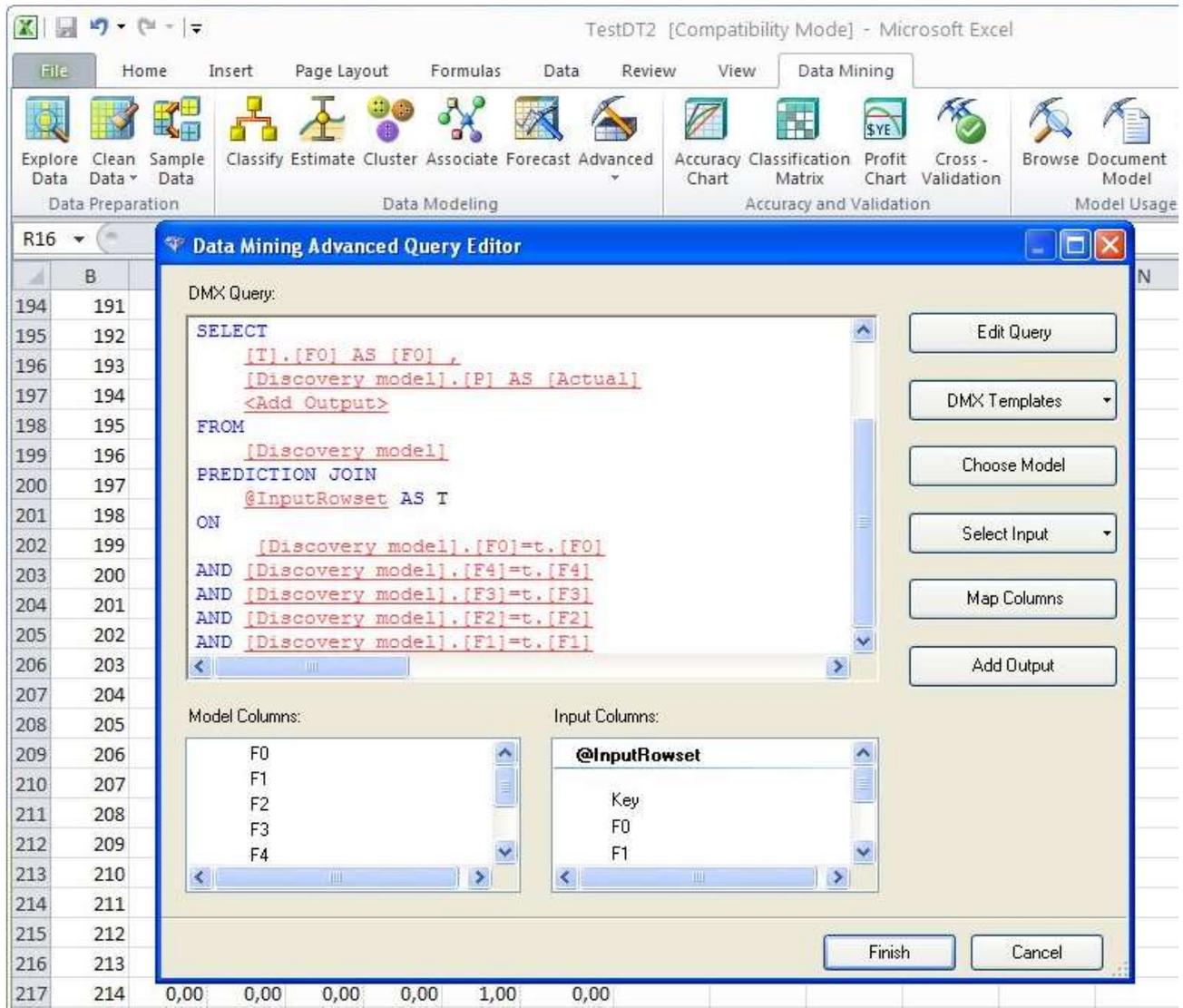


Рис. 14. Создание запроса на предсказание в Excel.

На данный момент Excel является самым удобным способом анализировать данные с помощью плагина Discovery.

7. Теоретическое сравнение

В силу определения семантического вероятностного вывода, который реализован системой «Discovery», в посылку правила всегда добавляются только такие предикаты, которые статистически значимо строго увеличивают условную вероятность правила. Такая проверка в других методах не проводится. В этом случае в правила могут включаться признаки, которые не имеют отношения к предсказанию и в том числе случайные. Опора на случайные признаки в предсказании может значительно ухудшить его точность.

Важность отсева случайных признаков критична, если нам надо не просто получить предсказание, а еще и проинтерпретировать полученные результаты. Специалист предметной области, интерпретируя правило, должен быть уверен, что признаки, входящие в правило действительно имеют отношение к предсказываемому признаку. Иначе интерпретация будет невозможна, либо специалист скажет, что правила бессмысленны и полученные предсказания – гадание на кофейной гуще и отчасти будет прав.

В Decision Trees для выбора разделяющих признаков используется энтропия, но на малых данных или в концах веток дерева могут включаться признаки, которые случайны и минимум энтропии признака получается чисто случайно. В Decision Trees нет статистического критерия проверки случайности добавляемых признаков. Поэтому правила, получаемые в Decision Trees, могут содержать случайные признаки, не имеющие отношение к предсказываемому признаку.

Хотя система «Discovery» и алгоритм Microsoft Association Rules достаточно похожи, в виду того, что в обоих подходах закономерности представляются в форме логических правил, тем не менее, между ними существуют принципиальные отличия.

1) В детерминированном случае, когда нет шума в данных, система «Discovery» обнаружит одно правило $A \& B \Rightarrow C$, истинное на данных. В то же время алгоритм, обнаруживающий ассоциативные правила, обнаружит все правила вида $A \& B \& \dots \& D \Rightarrow C$, которые получаются из правила $A \& B \Rightarrow C$ добавлением дополнительных условий D, F, \dots : $A \& B \& D \Rightarrow C$, $A \& B \& F \Rightarrow C$;

Например, при анализе тестовой таблицы 1 (описанной в приложении 1), где в качестве входных колонок использовались F_1, F_2, F_3 , а также колонка R_1 со случайными данными, алгоритмом Association Rules было обнаружено правило $(F_1 = 1 \wedge F_2 = 1 \wedge F_3 = 1) \Rightarrow P = 1$ с $UB = 1$, а также следующие 2 правила с $UB = 1$:

$$(F_1 = 1 \wedge F_2 = 1 \wedge F_3 = 1 \wedge R_1 = 1) \Rightarrow P = 1,$$

$$(F_1 = 1 \wedge F_2 = 1 \wedge F_3 = 1 \wedge R_1 = 0) \Rightarrow P = 1.$$

Таким образом, в случае, когда цель анализа – найти закономерности в данных, эксперт, использующий алгоритм Association Rules, получит три противоречивых правила с $УВ = 1$. Доверия к таким результатам не будет.

Кроме того, алгоритмом Association Rules было обнаружено множество правил следующего вида:

$$\begin{aligned} (F_1 = 1 \wedge R_1 = 1) &\Rightarrow P = 1, & (F_1 = 1 \wedge R_1 = 0) &\Rightarrow P = 1, \\ (F_2 = 1 \wedge R_1 = 1) &\Rightarrow P = 1, & (F_2 = 1 \wedge R_1 = 0) &\Rightarrow P = 1, \\ (F_1 = 1 \wedge F_2 = 1 \wedge R_1 = 1) &\Rightarrow P = 1, & (F_1 = 1 \wedge F_2 = 1 \wedge R_1 = 0) &\Rightarrow P = 1. \end{aligned}$$

Последние правила могут иметь приоритет над правилами с целевым предикатом $P = 0$, и, соответственно, ложно предсказывать 1, когда колонка P содержит 0. Например, в случае, когда на вход подаются колонки F_1, F_2, F_3 и только одна колонка со случайными данными R_1 , процент правильно предсказанных алгоритмом Association Rules значений составит около 87%, когда на вход подаются F_1, F_2, F_3 плюс две колонки R_1 и R_2 , точность падает до 70% (подробнее см. параграф 8.1, таб. 3).

Система «Discovery» в данном случае обнаружила только следующие правила:

$$\begin{aligned} (F_1 = 1 \wedge F_2 = 1 \wedge F_3 = 1) &\Rightarrow P = 1, & (F_1 = 0) &\Rightarrow P = 0, \\ (F_2 = 0) &\Rightarrow P = 0, & (F_3 = 0) &\Rightarrow P = 0, \end{aligned}$$

т.к. они уже имеют $УВ = 1$, и добавление каких-либо предикатов в условную часть правила не может увеличить условную вероятность правила. Предсказание, основанное на этих правилах, будет 100% точным.

2) когда есть шум в данных, система «Discovery» может обнаружить одно правило $A \& B \Rightarrow C$, представляющее собой вероятностный закон с определенным уровнем статистической значимости. В то же время алгоритм, обнаруживающий ассоциативные правила, должен обнаружить все детерминированные правила вида $A \& B \& D \Rightarrow C$, $A \& B \& F \Rightarrow C$, включающие случайные признаки, что приведет к ухудшению предсказания.

Например, при анализе тестовой таблицы 1 с 3% шумом, и колонками F_1, F_2, F_3 и R_1 в качестве входных колонок, системой «Discovery» были обнаружены только 4 правила, являющиеся СВЗ:

$$\begin{aligned} (F_1 = 1 \wedge F_2 = 1 \wedge F_3 = 1) &\Rightarrow P = 1, & (F_1 = 0) &\Rightarrow P = 0, \\ (F_2 = 0) &\Rightarrow P = 0, & (F_3 = 0) &\Rightarrow P = 0. \end{aligned}$$

Эти правила не содержат колонку со случайными данными, т.к. любое правило, имеющее в условной части колонку R_1 не пройдет проверку критерием Юла-Фишера и будет удалено.

Алгоритм Association Rules в данном примере обнаружил правило $(F_1 = 1 \wedge F_2 = 1 \wedge F_3 = 1) \Rightarrow P = 1$ с $УВ = p$, а также правила:

$$(F_1 = 1 \wedge F_2 = 1 \wedge F_3 = 1 \wedge R_1 = 1) \Rightarrow P = 1, (F_1 = 1 \wedge F_2 = 1 \wedge F_3 = 1 \wedge R_1 = 0) \Rightarrow P = 1,$$

причем одно из них имеет $UB > p$. Таким образом, в случае, когда цель анализа – найти закономерности в данных, эксперт, использующий алгоритм Association Rules, получит три противоречивых правила. При этом правило с наибольшей UB может содержать колонку со случайными данными.

Кроме того, из-за шума в данных алгоритм Association Rules обнаруживает множество правил следующего вида:

$$(F_1 = 1 \wedge F_2 = 1 \wedge F_3 = 0 \wedge R_1 = 1) \Rightarrow P = 1, (F_1 = 1 \wedge F_2 = 1 \wedge F_3 = 0 \wedge R_1 = 0) \Rightarrow P = 1,$$

$$(F_1 = 1 \wedge F_2 = 0 \wedge F_3 = 1 \wedge R_1 = 1) \Rightarrow P = 1, (F_1 = 1 \wedge F_2 = 0 \wedge F_3 = 1 \wedge R_1 = 0) \Rightarrow P = 1,$$

$$(F_1 = 0 \wedge F_2 = 1 \wedge F_3 = 1 \wedge R_1 = 1) \Rightarrow P = 1, (F_1 = 0 \wedge F_2 = 1 \wedge F_3 = 1 \wedge R_1 = 0) \Rightarrow P = 1,$$

которые могут иметь приоритет над правилами с целевым предикатом $P = 0$ и ложно предсказывать 1, когда колонка P содержит 0. Это приводит к ухудшению предсказания (что показано на рис. 19 в приложении 5).

Сравнивая алгоритм «Discovery» с Neural Network, нужно еще раз отметить, что система «Discovery» имеет четкий статистический критерий, позволяющий исключать случайные признаки из правил. Алгоритм Neural Network такого критерия не имеет. Его метод остановки обучения связан с точностью предсказания на обучающем наборе, из-за чего этот алгоритм склонен к переобучению (overfitting). Если на обучающем наборе подбирать параметры алгоритма, например, увеличивать кол-во скрытых слоев, можно получить очень хорошие результаты предсказания на обучающем наборе, но на тестовом наборе в таком случае предсказание будет неудовлетворительным.

8. Экспериментальное сравнение

8.1. Сравнение Discovery с Association Rules на модельных данных

Key	F1	F2	F3	P	R1
1	1	1	1	1	0
2	1	1	1	1	1
3	1	1	0	0	0
4	1	1	0	0	0
5	1	0	1	0	1
6	1	0	1	0	0
7	1	0	0	0	1
8	1	0	0	0	1
9	0	1	1	0	1
10	0	1	1	0	0
11	0	1	0	0	1
12	0	1	0	0	0
13	0	0	1	0	0
14	0	0	1	0	1
15	0	0	0	0	0
16	0	0	0	0	1

Для обучения алгоритмов Discovery и Association Rules использовались: Тестовая таблица 1, полученная из приведенной слева таблицы, масштабированием в несколько раз, и добавлением колонок $R_1 - R_5$ со случайными данными; а также Тестовая таблица 2, которая построена по такому же принципу, только её колонки F_1, F_2, F_3 содержат значения 1,2,3, и, соответственно, её длина кратна 27 (формальное описание таблиц находится в приложении 1)

Заметим, что обе таблицы содержат две закономерности, достаточные для точного предсказания:

$$(F_1 = 1) \& (F_2 = 1) \& (F_3 = 1) \rightarrow (P = 1)$$

$$(F_1 \neq 1) | (F_2 \neq 1) | (F_3 \neq 1) \rightarrow (P = 0)$$

Т.к. оба сравниваемых алгоритма не могут содержать в своих правилах отрицание и логическое «или», вторая закономерность находится в виде нескольких правил, предсказывающих $P = 0$.

Для анализа данных таблиц применим систему «Discovery». В качестве входных колонок используем колонки $F_1, F_2, F_3, R_1 - R_5$, в качестве целевого признака – колонку P. В качестве критериев статистической значимости используемая реализация применяет точный критерий Фишера с пороговым значением 0,05 и критерий Юла с пороговым значением 0,1.

Система «Discovery» обнаруживает следующие правила с условной вероятностью (УВ) равной 1.

На тестовой таблице 1:

УВ 1: IF (F1 = 1) AND (F2 = 1) AND (F3 = 1) THEN (P = 1);

УВ 1: IF (F3 = 0) THEN (P = 0);

УВ 1: IF (F2 = 0) THEN (P = 0);

УВ 1: IF (F1 = 0) THEN (P = 0);

На тестовой таблице 2:

УВ 1: IF (F1 = 1) AND (F2 = 1) AND (F3 = 1) THEN (P = 1);

УВ 1: IF (F3 = 2) THEN (P = 0);

УВ 1: IF (F3 = 3) THEN (P = 0);

УВ 1: IF (F2 = 2) THEN (P = 0);

УВ 1: IF (F2 = 3) THEN (P = 0);

УВ 1: IF ($F_1 = 2$) THEN ($P = 0$);

УВ 1: IF ($F_1 = 3$) THEN ($P = 0$);

Теперь проанализируем тестовые таблицы с помощью Association Rules. В качестве входных колонок используем колонки F_1, F_2, F_3 , в качестве целевого признака – колонку P .

Алгоритм Association Rules обнаруживает 20 правил с УВ, равной 1, на тестовой таблице 1 (57 на тестовой таблице 2), в том числе и все правила найденные системой «Discovery». При добавлении колонки R_1 ко входным колонкам Association Rules обнаруживает уже 60 правил с УВ, равной 1, на тестовой таблице 1 (228 на тестовой таблице 2). В Приложении 2 на рисунке 15 показано как растет количество правил с УВ, равной 1, обнаруженных алгоритмом Association Rules, при добавлении к F_1, F_2, F_3 колонок $R_1 - R_5$ в качестве входных колонок.

Далее посмотрим, как добавление в модель колонок со случайными данными ухудшает качество предсказания «Ассоциативных правил», а также покажем, что количество записей в таблице незначительно влияет на качество предсказания.

Таблица 3. Процент правильно предсказанных значений на тестовой таблице 1.

	0	1	2	3	4	5
Association Rules, n = 256	100%	87.5%	69.53%	45.70%	29.29%	19.53%
Association Rules, n = 1024	100%	87.59%	70.41%	45.31%	30.56%	23.92%
Association Rules, n = 4096	100%	88.15%	69.14%	47.07%	32.86%	24.43%

Таблица 4. Процент правильно предсказанных значений на тестовой таблице 2.

	0	1	2	3	4	5
Association Rules, n = 243	100%	91.81%	74.16%	58.46%	25.68%	16.26%
Association Rules, n = 2187	100%	91.95%	75.76%	55.05%	28.30%	17.92%
Association Rules, n = 19683	100%	91.06%	76.85%	55.19%	29.71%	18.46%

Отметим, что на тестовых таблицах 1 и 2 без шума система «Discovery» имеет 100% правильно предсказанных значений целевой колонки P при любом количестве случайных колонок $R_1 - R_5$, участвующих в обучении модели.

В Приложении 3 на рис. 16, 17 проводится сравнение качества предсказания алгоритма Association Rules и системы «Discovery».

Рассмотрим тестовые таблицы 1 и 2 с наложением 3% шума. В качестве входных колонок используем колонки $F_1, F_2, F_3, R_1 - R_5$, в качестве целевого признака – колонку P .

В результате система «Discovery» обнаруживает следующие правила, являющиеся СВЗ.

На тестовой таблице 1:

УВ 0,854: IF (F1 = 1) AND (F2 = 1) AND (F3 = 1) THEN (P = 1)

УВ 0,959: IF (F2 = 0) THEN (P = 0)

УВ 0,944: IF (F1 = 0) THEN (P = 0)

УВ 0,945: IF (F3 = 0) THEN (P = 0)

Первые два из них являются МСЗ.

На тестовой таблице 2:

УВ 0,910: IF (F1 = 1) AND (F2 = 1) AND (F3 = 1) THEN (P = 1)

УВ 0,988: IF (F1 = 2) AND (F2 = 3) AND (F3 = 2) THEN (P = 0)

УВ 0,962: IF (F2 = 2) AND (F3 = 3) THEN (P = 0)

УВ 0,967: IF (F1 = 3) AND (F2 = 2) THEN (P = 0)

УВ 0,971: IF (F1 = 3) AND (F3 = 3) THEN (P = 0)

УВ 0,977: IF (F1 = 3) AND (F2 = 3) THEN (P = 0)

Первые два из них также являются МСЗ.

Проанализируем тестовые таблицы 1 и 2 с наложением 3% шума с помощью Association Rules. В качестве входных колонок используем колонки F_1, F_2, F_3 , в качестве целевого признака – колонку P . В результате, на тестовой таблице 1 Association Rules обнаруживает 29 правил, из них 20 правил с УВ > 0.85, в том числе и все правила, найденные системой «Discovery». На тестовой таблице 2 Association Rules обнаруживает 60 правил с УВ > 0.85, в том числе и все правила, найденные системой «Discovery». В Приложении 4 на рис. 18 показано, как растет количество правил с УВ > 0.85, обнаруженных алгоритмом Association Rules, при добавлении колонок $R_1 - R_5$ в качестве входных колонок.

В Приложении 5 проводится сравнение качества предсказания алгоритма Association Rules и системы Discovery на тестовой таблице 2 с шумом 0%, 2% и 3%.

8.2. Сравнение Discovery с Decision Trees и Neural Network на модельных данных.

В качестве анализируемых данных используются таблицы следующего вида:

F0	F1	F2	F3	F4	P	R1
1	1	0	0	0	1	0
1	1	1	0	0	1	1
1	1	1	1	0	1	0
0	1	1	1	1	1	0
0	0	1	1	1	1	1
0	0	0	1	1	1	0
1	0	1	0	0	0	1
1	0	1	0	0	0	1
1	0	1	0	0	0	0
1	0	0	1	0	0	1
1	0	0	1	0	0	0
0	1	0	1	0	0	0
0	1	0	1	0	0	1
0	1	0	1	0	0	0
0	1	0	0	1	0	1
0	1	0	0	1	0	0
0	0	1	0	1	0	1
0	0	1	0	1	0	1
0	0	1	0	1	0	0
0	0	0	1	0	0	1
0	0	0	1	0	0	0
0	0	0	0	1	0	1

Для обучения сравниваемых алгоритмов использовалась Тестовая таблица 3, полученная из таблицы, приведенной слева, масштабированием в 8 раз (и удалением нескольких записей в конце таблицы), так, что итоговая длина Тестовой таблицы 3 равна 197.

В качестве входных колонок использовались колонки $F_0 - F_4$ и $R_1 - R_5$, в качестве целевого признака – колонка P .

Данная тестовая таблица содержит следующие закономерности, достаточные для точного предсказания:

$$(F_0 = 1) \& (F_1 = 1) \rightarrow (P = 1);$$

$$(F_1 = 1) \& (F_2 = 1) \rightarrow (P = 1);$$

$$(F_2 = 1) \& (F_3 = 1) \rightarrow (P = 1);$$

$$(F_3 = 1) \& (F_4 = 1) \rightarrow (P = 1);$$

А также несколько закономерностей, предсказывающие $P = 0$.

Сначала для анализа таблицы применим систему «Discovery». В качестве входных колонок используем колонки $F_0 - F_4$, в качестве целевого признака – колонку P . В качестве критериев статистической значимости применим точный критерий Фишера с пороговым значением 0,13 и критерий Юла с пороговым значе-

нием минус 0,03.

Система «Discovery» обнаруживает следующие правила с условной вероятностью (УВ) равной 1, предсказывающие $P = 1$:

УВ 1: IF (F0 = 1) AND (F1 = 1) THEN (P = 1);

УВ 1: IF (F1 = 1) AND (F2 = 1) THEN (P = 1);

УВ 1: IF (F2 = 1) AND (F3 = 1) THEN (P = 1);

УВ 1: IF (F3 = 1) AND (F4 = 1) THEN (P = 1).

А также несколько правил, предсказывающие $P = 0$. Обнаруженные правила являются достаточными для точного предсказания.

Далее проанализируем Тестовую таблицу 3 с помощью Decision Trees. В качестве входных колонок используем колонки $F_0 - F_4$, в качестве целевой – колонку P. Параметр COMPLEXITY_PENALTY для Decision Trees установим равным 0.001. Алгоритм Decision Trees обнаружил следующие правила, предсказывающие $P = 1$:

УВ 0.965: IF (F0 = 1) AND (F1 = 1) THEN (P = 1);

УВ 0.982: IF (F1 = 0) AND (F3 = 1) AND (F4 = 1) THEN (P = 1);

А также несколько правил, предсказывающие $P = 0$. Нетрудно заметить, что найденных Decision Trees правил не достаточно для точного предсказания $P = 1$.

Алгоритм Neural Network не находит закономерности в виде логических правил, потому сложно сделать вывод о точности предсказания не проводя собственно теста.

Для сравнения качества предсказания алгоритмов будем использовать тестовую таблицу 4, полученную масштабированием тестовой таблицы 3 в 4 раза. Введем следующую меру ошибки: если алгоритм ошибочно предсказывает 1 вместо 0, то стоимость ошибки равна 1, если ошибочно предсказывает 0 вместо 1, то стоимость ошибки равна 3. Данную меру ошибки можно представить в виде таблицы:

Actual\Predicted	0	1
0	0	1
1	3	0

Максимальная ошибка, возможная на тестовой таблице 4 имеет стоимость 1172. В результате выполнения предсказания для записей тестовой таблицы 4, Discovery не допустила ни одной ошибки, т.е. предсказала абсолютно точно.

Decision Trees на тестовой таблице 4 допустила ошибку общей стоимостью 96, или 8,2% от максимальной стоимости ошибки.

Проанализируем, как добавление шума в таблицы влияет на обнаруживаемые закономерности и ухудшает качество предсказания алгоритмов Discovery и Decision Trees.

Под таблицей с n-процентным шумом мы подразумеваем таблицу, в которой в n процентах ячеек, выбранных случайным образом, значение признака заменено на противоположное.

На тестовой таблице 3 с шумом 1% система «Discovery» обнаруживает следующие правила, предсказывающие $P = 1$:

УВ 0.958: IF (F0 = 1) AND (F1 = 1) THEN (P = 1);

УВ 0.962: IF (F1 = 1) AND (F2 = 1) THEN (P = 1);

УВ 0.960: IF (F2 = 1) AND (F3 = 1) THEN (P = 1);

УВ 1.000: IF (F3 = 1) AND (F4 = 1) THEN (P = 1);

А также правила, предсказывающие $P = 0$. На этой же таблице Decision Trees обнаруживает следующие правила, предсказывающие $P = 1$:

УВ 0.982: IF (F1 = 0) AND (F3 = 1) AND (F4 = 1) THEN (P = 1);

УВ 0.982: IF (F0 = 1) AND (F1 = 1) AND (F2 = 1) THEN (P = 1);

УВ 0.878: IF (F0 = 0) AND (F1 = 1) AND (F2 = 1) THEN (P = 1);

А также несколько правил, предсказывающие $P = 0$. В следующей таблице показаны результаты предсказаний, выполненных алгоритмами Discovery и Decision Trees. Алгоритмы обучались на таблице 3 с шумами в 2, 4 и 6 процента. Предсказание выполнялось для записей таблицы 4 с теми же шумами 2, 4, 6 процента.

Таблица 5. Абсолютная и относительная стоимость ошибок сравниваемых алгоритмов на тестовой таблице три с шумами в 2, 4 и 6 процентов.

	Discovery		Decision Trees		Neural Network	
	Стоимость	% от max	Стоимость	% от max	Стоимость	% от max
шум 0%	0	0	96	8.2	0	0
шум 2%	70	6	162	13.8	203	17.3
шум 4%	136	11.6	134	11.4	238	20.3
шум 6%	190	16.2	255	21.8	354	30.2

Во всех приведенных случаях система Discovery работала лучше Decision Trees и Neural Network.

8.3. Сравнение алгоритмов на реальных данных. Таблица "Statlog German Credit Data".

Для следующего теста возьмем данные с известного DM-репозитория UC Irvine Machine Learning Repository (<http://archive.ics.uci.edu/ml/>). Таблица "Statlog German Credit Data" содержит данные людей, претендующих на получение заемных средств в банке [9]. Цель интеллектуального анализа таблицы - определить кредитные риски претендента. Таблица размером 1000 записей, содержит 20 входных атрибутов, таких как: кредитная история, цель кредита, размер кредита, накопления на счетах, время занятости на текущем месте работы, пол/семейное положение, наличие недвижимости и т.д. Предсказываемый атрибут принимает два значения: 1 - низкие кредитные риски, 2 - высокие риски.

К данным прилагается матрица стоимостей ошибок, в которой совершенно логично ука-

Actual\Predicted	1	2
1	0	1
2	5	0

зано, что ложное предсказание низких кредитных рисков наносит больший ущерб, чем ложное предсказание высоких рисков. Максимальная возможная ошибка имеет стоимость 2200.

Для устранения искажений результатов теста вследствие эффекта переобучения (overfitting) тестируемых DM-алгоритмов, воспользуемся методом скользящего экзамена [4]: исходная таблица A случайным образом разбивается на n равных по размеру, не пересекающихся наборов $A(i)$, $i = 0 \dots n-1$. В нашем случае $n = 20$. Формируются наборы $B(i)$ как дополнения к

$A(i)$, т.е. $B(i) = A - A(i)$. Затем проводится n тестов, в которых DM-модель обучается на $B(i)$, а предсказание проводится для записей из $A(i)$. Результаты предсказаний по всем $A(i)$ соединяются в одну таблицу, для которой считается общая стоимость ошибки.

Проведем интеллектуальный анализ таблицы German Credit Data с помощью описанного выше метода алгоритмами Discovery, Decision Trees, Neural Network и Association Rules.

Для Discovery в качестве критериев статистической значимости применим точный критерий Фишера с пороговым значением 0,06 и критерий Юла с пороговым значением 0. В качестве меры правила возьмем величину Юла. Для Decision Trees установим параметр COMPLEXITY_PENALTY равным 0,01. Для Neural Network установим параметр HOLDOUT_PERCENTAGE равным 40 и параметр HIDDEN_NODE_RATIO равным 6. Данные параметры обеспечивают наилучшие результаты алгоритмов.

Таблица 6. Абсолютная и относительная стоимость ошибок сравниваемых алгоритмов на таблице German Credit Data.

	Discovery		Decision Trees		Neural Network	
	Стоимость	% от max	Стоимость	% от max	Стоимость	% от max
German Credit Data	700	31.8	724	32.9	936	42.5

Как видим, система Discovery в данном тесте показала несколько лучшие результаты, чем Decision Trees, и значительно лучшие, чем Neural Network. Association Rules на данном тесте показал неприемлемо плохие результаты: на всех записях тестовых таблиц алгоритм предсказал значение 1, и ни разу не предсказал значение 2.

8.4. Сравнение алгоритмов на реальных данных. Таблица "Magic".

Следующий тест также взят с DM-репозитория UC Irvine Machine Learning Repository. Таблица «Magic» содержит данные телескопа Черенкова, ведущего наблюдение за гамма-лучами высокой энергии [8]. Цель интеллектуального анализа таблицы – отделять истинные (сигнальные) высокоэнергетические фотоны от фоновых фотонов, инициированных космическим излучением в верхних слоях атмосферы. Таблица содержит 19020 записей, 10 атрибутов, содержащие параметры излучения, а также целевой (предсказываемый) атрибут, принимающий 2 значения: 1 – сигнальный фотон, 2 – фоновый фотон. Каждая запись таблицы соответствует одному фотону, зарегистрированному телескопом.

К данным не прилагается таблица стоимостей ошибок, потому определим стоимости ошибок обоих типов равными 1. Тогда максимальная возможная ошибка имеет стоимость 19020.

Actual\Predicted	1	2
1	0	1
2	1	0

Для устранения искажений результатов теста из-за эффекта переобучения (overfitting) тестируемых ДМ-алгоритмов, используем тот же прием, который был описан выше. А именно: исходная таблица «Magic» разбивается на 20 пар таблиц $\langle M(i); M-M(i) \rangle$, где $M-M(i)$ используется для обучения, а на $M(i)$ производится предсказание. Т.е. обучение и последующее тестирование производятся на разных наборах данных.

Проведем интеллектуальный анализ таблицы «Magic» с помощью описанного выше метода алгоритмами Discovery, Decision Trees, Neural Network и Association Rules..

Для Discovery в качестве критериев статистической значимости применим точный критерий Фишера с пороговым значением 0.08 и критерий Юла с пороговым значением 0.1, условную вероятность правил ограничим числом 0.8. В качестве меры правила используем условную вероятность. Для Association Rules установим MINIMUM_IMPORTANCE равной 0.2 и MINIMUM_PROBABILITY равной 0.8. Для алгоритмов Decision Trees и Neural Network оставим параметры по умолчанию. Данные параметры обеспечивают наилучшие результаты алгоритмов на данной таблице.

Таблица 7. Абсолютная и относительная стоимость ошибок алгоритмов на таблице Magic.

	Стоимость	% от max
Discovery	3692	19.41
Decision Trees	3336	17.54
Neural Network	3946	20.75
Association Rules	3964	20.84

Как видно из таблицы, данный тест оказался особенно удобен для алгоритма Decision Trees, который показал здесь наилучшие результаты. Система Discovery показала лучшие результаты, чем Neural Network и Association Rules.

9. Заключение

В данной работе приведено сравнение системы «Discovery» с алгоритмами Microsoft Association Rules, Decision Trees и Neural Network, встроенными в Microsoft SQL Server Analysis Services (SSAS). Для проведения сравнения система «Discovery» была реализована в виде плагина SSAS, что позволило использовать единую среду разработки Business Intelligence Development Studio, единые средства визуализации Data Mining моделей, и стандартные средства сравнения качества Data Mining моделей.

Для сравнения DM-алгоритмов автором был также разработан ряд модельных тестов, на которых теоретически система «Discovery» должна работать лучше, чем другие рассматриваемые алгоритмы. Также были проведены эксперименты на этих модельных данных, показывающие, что и практически на этих данных система «Discovery» работает лучше. Эти эксперименты показывают, что система «Discovery» имеет свою область применимости. Для сравнения системы «Discovery» с другими методами на произвольных реальных данных были использованы данные из известного репозитория UCI Machine Learning Repository. На этих данных система «Discovery» работает не хуже других методов. В целом эти результаты показывают, что реализация системы «Discovery» в виде плагина к службам Microsoft SSAS вполне оправдана. Более того, подключение системы «Discovery» в качестве плагина к Microsoft Excel, как и другие возможности по созданию «тонких» клиентских приложений, использующих «Discovery», делают эту систему доступной максимально широкому кругу пользователей.

Приложение

Приложение 1

В качестве модельных данных используются следующие тестовые таблицы. Тестовая таблица 1 имеет три значимых колонки F_1, F_2, F_3 определяемые следующим выражением:

$$F_1(i) = \begin{cases} 1, i \in (1, 512k) \\ 0, i \in (512k+1, 1024k) \end{cases}, 1024 - \text{количество записей в таблице.}$$

$$F_2(i) = \begin{cases} 1, i \in (1, 256k) \cup (512k+1, 768k) \\ 0, i \in (256k+1, 512k) \cup (768k+1, 1024k) \end{cases}$$

$$F_3(i) = \begin{cases} 1, i \in (1, 128k) \cup (256k+1, 384k) \cup (512k+1, 640k) \cup (768k+1, 896k) \\ 0, i \in (128k+1, 256k) \cup (384k+1, 512k) \cup (640k+1, 768k) \cup (896k, 1024k) \end{cases}$$

и колонку P , используемую в качестве целевого признака:

$$P(i) = \begin{cases} 1, i \in (1, 128k) \\ 0, i \in (128k+1, 1024k) \end{cases},$$

а также 5 колонок $R_1 - R_5$ с независимыми Бернуллиевскими случайными значениями.

Тестовая таблица 2 также имеет три значимых колонки F_1, F_2, F_3 определяемые следующим выражением:

$$F_1(i) = \begin{cases} 1, i \in (1, 729k) \\ 2, i \in (729k+1, 1458k) \\ 3, i \in (1458k+1, 2187k) \end{cases}$$

$$F_2(i) = \begin{cases} 1, i \in (1, 243k) \cup (729k+1, 972k) \cup (1458k+1, 1701k) \\ 2, i \in (243k+1, 486k) \cup (972k+1, 1215k) \cup (1701k+1, 1944k) \\ 3, i \in (486k+1, 729k) \cup (1215k+1, 1458k) \cup (1944k+1, 2187k) \end{cases}$$

$$F_3(i) = \begin{cases} 1, i \in \bigcup_{j=0}^2 \left((1+729kj, 81k+729kj) \cup (1+243k+729kj, 324k+729kj) \right) \\ \quad \cup (1+486k+729kj, 567k+729kj) \\ 2, i \in \bigcup_{j=0}^2 \left((1+81k+729kj, 162k+729kj) \cup (1+324k+729kj, 405k+729kj) \right) \\ \quad \cup (1+567k+729kj, 648k+729kj) \\ 3, i \in \bigcup_{j=0}^2 \left((1+162k+729kj, 243k+729kj) \cup (1+405k+729kj, 486k+729kj) \right) \\ \quad \cup (1+648k+729kj, 729k+729kj) \end{cases}$$

(2187 – количество записей в таблице) и колонку P , используемую в качестве целевого признака:

$$P(i) = \begin{cases} 1, i \in (1, 81k) \\ 0, i \in (1+81k, 2187k) \end{cases},$$

а также 5 колонок $R_1 - R_5$ с независимыми Бернуллиевскими случайными значениями.

На тестовых таблицах можно увидеть две простые закономерности:

$$(F_1 = 1) \& (F_2 = 1) \& (F_3 = 1) \rightarrow (P = 1)$$

$$(F_1 \neq 1) | (F_2 \neq 1) | (F_3 \neq 1) \rightarrow (P = 0)$$

Под тестовой таблицей с n -процентным шумом мы подразумеваем тестовую таблицу, в которой в n процентах ячеек, выбранных случайным образом, значение заменено на противоположное.

Приложение 2

На следующем графике показано, как растет количество правил с УВ, равной 1, обнаруженных алгоритмом Association Rules, при добавлении к F_1, F_2, F_3 колонок $R_1 - R_5$ в качестве входных колонок. Здесь и далее на горизонтальной оси отмечено количество колонок $R_1 - R_5$ со случайными данными, используемых (в дополнение к F_1, F_2, F_3) в качестве входных данных.

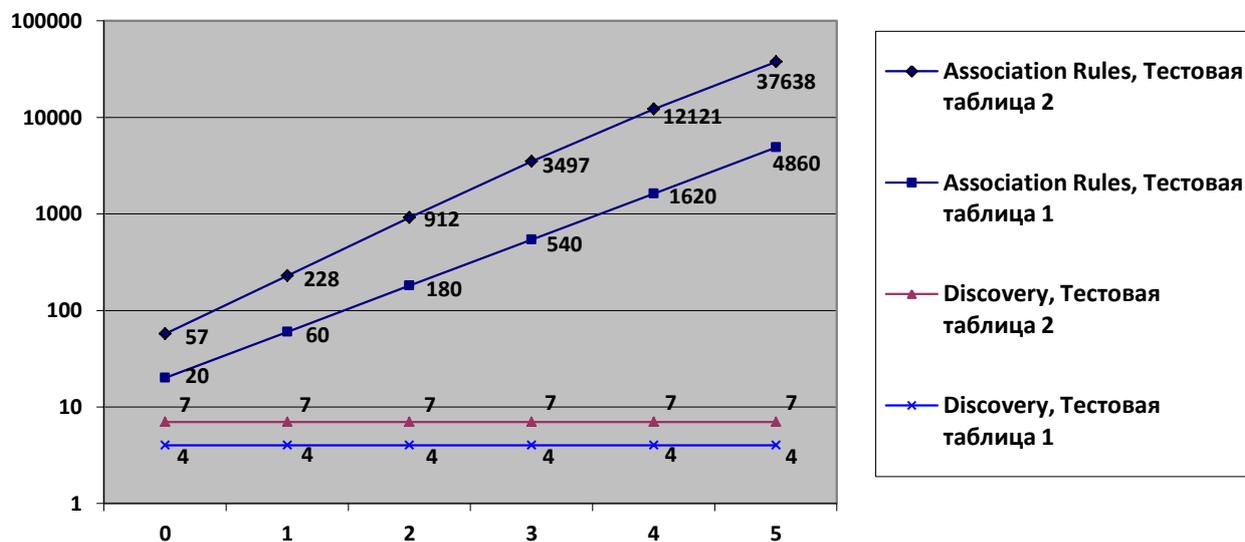


Рис. 15. Количество правил с УВ = 1, обнаруженных алгоритмом Association Rules.

Как видим, количество правил, найденных Association Rules, экспоненциально растет при добавлении новых колонок.

Приложение 3

На следующем графике показан процент правильно предсказанных значений алгоритма Association Rules и системы «Discovery» в зависимости от количества колонок $R_1 - R_5$ со случайными данными, используемых в качестве входных данных, а также размера тестовой таблицы.

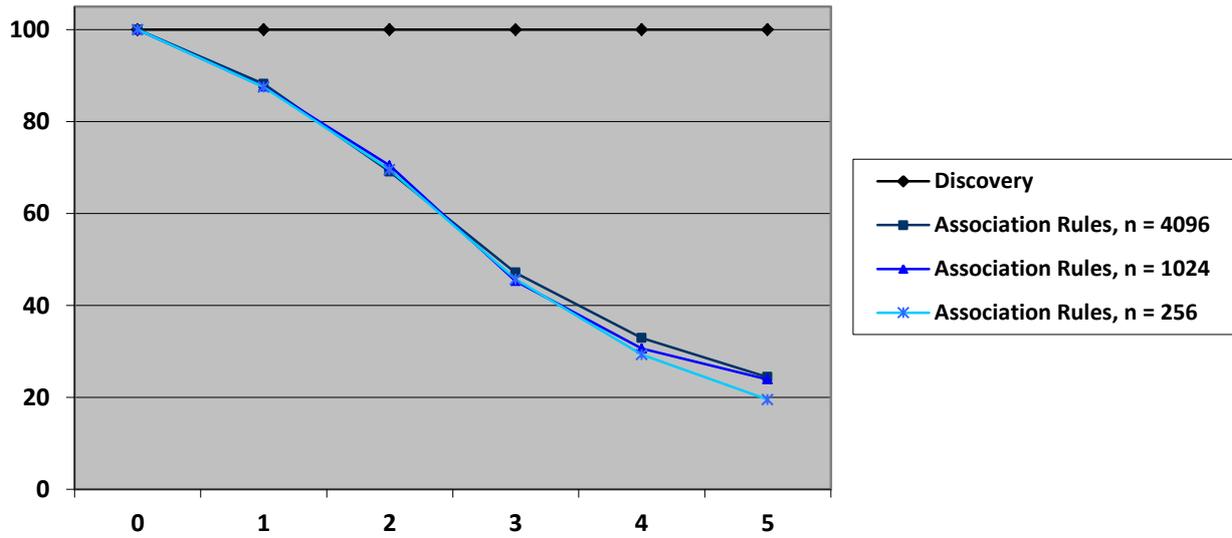


Рис. 16. Процент правильно предсказанных значений при анализе тестовой таблицы 1.

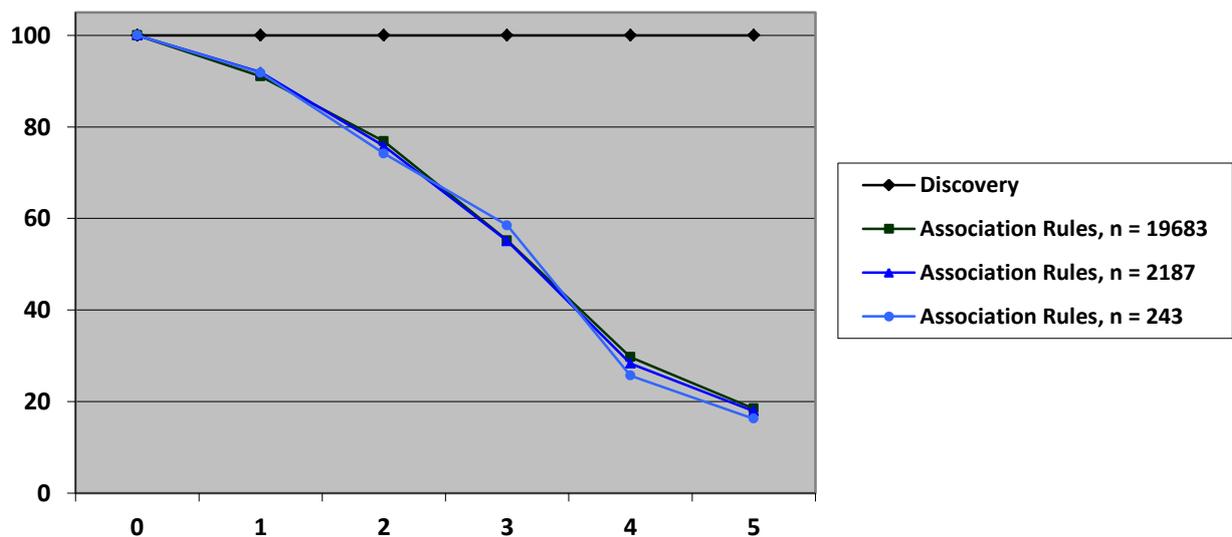


Рис. 17. Процент правильно предсказанных значений при анализе тестовой таблицы 2.

Как видим, при увеличении числа колонок $R_1 - R_5$ со случайными данными, используемых в качестве входных данных, качество предсказания алгоритма Association Rules значительно падает. Размер тестовой таблицы незначительно влияет на результат.

Приложение 4

На следующем графике показано, как растет количество правил с $UB > 0.85$, обнаруженных алгоритмом Association Rules, при добавлении колонок $R_1 - R_5$ в качестве входных колонок. Анализируются тестовые таблицы с наложением 3% шума.

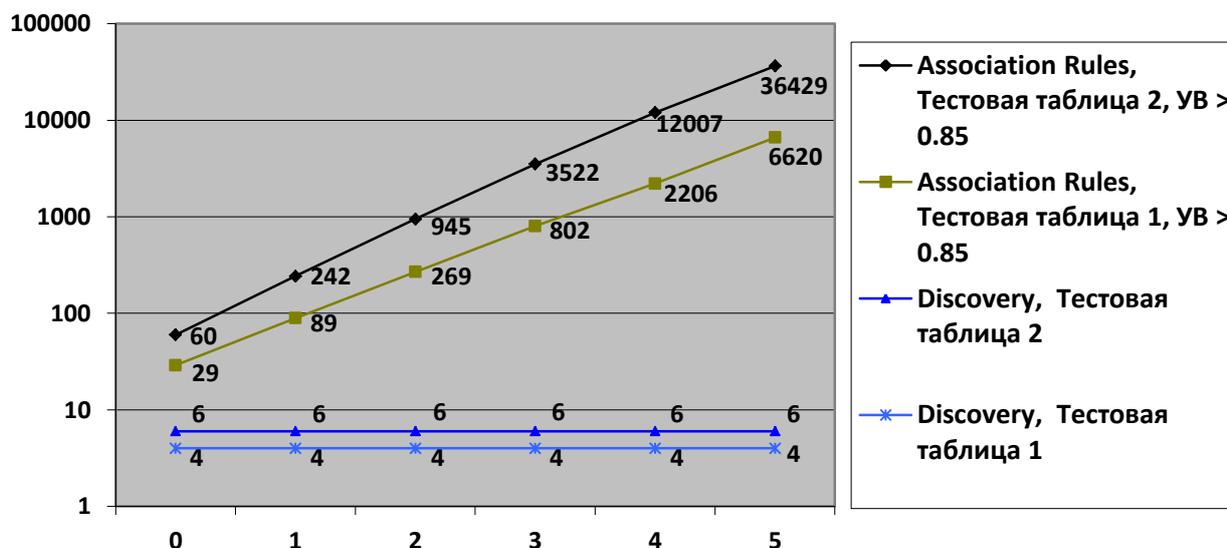


Рис. 18. Количество правил, обнаруженных алгоритмом Association Rules и системой Discovery.

Приложение 5

Сравним качество предсказания системы “Discovery” и алгоритма Association Rules на тестовой таблице 2 с шумом 0%, 2% и 3%. Как видим, система “Discovery” дает наиболее близкое к идеальному предсказание, причем качество прогнозирования не зависит от количества колонок со случайными данными, используемых в качестве входных данных, а зависит только от величины шума. Заметим, что модель, обученная с помощью алгоритма “Discovery” на данных с шумом, будет давать 100% верные предсказания на данных без шума. Алгоритм Association Rules дает аналогичное Discovery качество предсказания в случае, когда случайные колонки не участвуют в обучении модели. При добавлении в модель случайных колонок $R_1 - R_5$ качество прогнозирования Association Rules значительно падает.

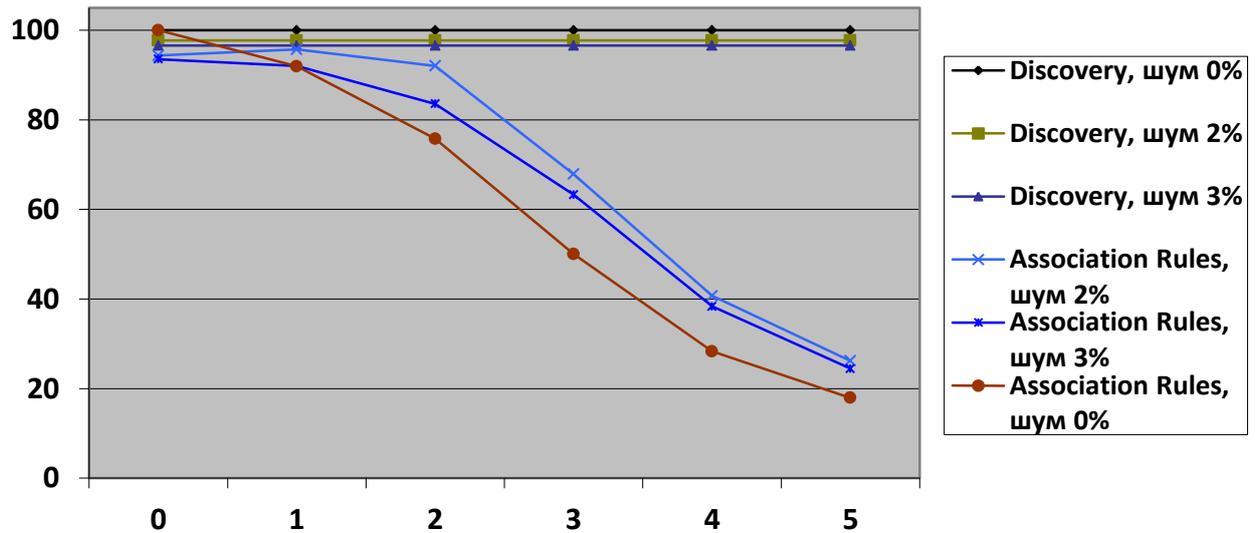


Рис. 19. Процент правильно предсказанных значений при анализе тестовой таблицы 2.

Заметим, что качество предсказания Association Rules на данных без шума при добавлении 2 и более колонок $R_1 - R_5$ заметно хуже, чем на данных с шумом 2% или 3%. Это объясняется тем, что на данных без шума Association Rules обнаруживает огромное количество «равноправных» правил с $UB = 1$, выбрать верное из которых не представляется возможным.

Список литературы

1. Витяев Е.Е. Извлечение знаний из данных. Компьютерное познание. Модели когнитивных процессов. – НГУ, Новосибирск, 2006. – 293 с.
2. Закс Ш. Теория статистических выводов – М.: Мир, 1975. – 776 с.
3. Кендалл М. Дж., Стьюарт А. Статистические выводы и связи – М.: Наука, 1973. – 899 с.
4. Лбов Г.С., Бериков В.Б. Устойчивость решающих функций в задачах распознавания образов и анализа разнотипной информации. – Новосибирск: ИМ СО РАН, 2005. – С. 133–135.
5. Data Mining Add-ins // Microsoft Office documentation. [Электронный ресурс].
URL: <http://office.microsoft.com/en-us/excel-help/data-mining-add-ins-NA010342915.aspx> (дата обращения: 31.08.2015).
6. Halpern J. Y. An analysis of first-order logic of probability. – Artificial Intelligence. 1990. – С. 311–350.
7. Kovalerchuk, B.Y., Vityaev E.E. Data mining in finance: Relational and hybrid methods. Kluwer, 2000. 308 p.
8. MAGIC Gamma Telescope Data Set // UCI Machine Learning Repository. [Электронный ресурс].
URL: <http://archive.ics.uci.edu/ml/datasets/MAGIC+Gamma+Telescope> (дата обращения: 31.08.2015).
9. Statlog (German Credit Data) Data Set // UCI Machine Learning Repository. [Электронный ресурс].
URL: [http://archive.ics.uci.edu/ml/datasets/Statlog+\(German+Credit+Data\)](http://archive.ics.uci.edu/ml/datasets/Statlog+(German+Credit+Data)) (дата обращения: 31.08.2015).
10. Tang Z., MacLennan J. Data Mining with SQL Server 2005. Wiley Publishing, Inc., 2005. 483 p.
11. Vityaev E., Kovalerchuk B. Empirical Theories Discovery based on the Measurement Theory // Mind and Machine, 2006. V.14. N4, P. 551-573.
12. Vityaev E. The logic of prediction. In: Mathematical Logic in Asia // Proceedings of the 9th Asian Logic Conference (August 16-19, 2005, Novosibirsk, Russia). World Scientific, Singapore, 2006. P. 263-276.

УДК 004.451.33

О параллельной разметке графов объектов

Щербина С.А. (Институт систем информатики СО РАН)

Михеев В.В. (Институт систем информатики СО РАН)

В статье рассматривается задача параллельной разметки графа объектов в рамках системы автоматического управления памятью. Авторы формулируют набор ограничений для решения данной задачи и строят алгоритм параллельной разметки, учитывающий эти ограничения. Предложенный алгоритм был реализован в Java-машине Excelsior RVM и апробирован на реальных Java-приложениях. Полученные результаты показывают значительное ускорение разметки графа объектов в большинстве случаев.

Ключевые слова: распределение памяти, сборка мусора, управляемые среды, производительность, параллельные алгоритмы, синхронизация

1. Введение

В реализациях современных языков программирования используется техника автоматического управления памятью, также называемая сборкой мусора[9]. Одной из известных проблем такого подхода является недостаточная производительность приложений, а также, зачастую, возникновение значительных пауз во время исполнения, вызванных работой сборщика мусора. С ростом объёмов используемой приложениями памяти эта проблема только усиливается.

Разметка графа объектов занимает центральное место в задаче сборки мусора. Самые тривиальные алгоритмы сборки мусора состоят именно в разметке, и поэтому её производительность определяет производительность всего сборщика мусора. На практике часто используются инкрементальные[1, 13] сборщики мусора. Несмотря на то, что их малые циклы сборки могут не столь сильно зависеть от производительности разметки, она занимает значительную долю во времени работы полных циклов сборки, то есть влияет на время самых длительных пауз в работе приложения, вызываемых сборщиком мусора. Время разметки актуально и для конкурентных (*concurrent*)[5] сборщиков мусора, которые могут выполняться параллельно с работой приложения, не вызывая вообще или вызывая лишь небольшие паузы в ней, но от длительности разметки графа объектов зависит производительность приложения[5].

С другой стороны, прогресс в росте вычислительной мощности компьютеров свернул с пути увеличения производительности одного процессора в сторону наращивания числа процессоров и ядер. Сочетание этих соображений и послужило мотивацией для выполнения данной работы. Иными словами, целью является исследование возможности применения параллельных алгоритмов для сокращения длительности разметки графа объектов и, как следствие, улучшения производительности и отзывчивости приложений, использующих автоматическое управление памятью.

Работа структурирована следующим образом. Раздел 2 описывает проблемы параллельной разметки. В разделе 3 конструируется и анализируется алгоритм, который призван эти проблемы решать. В разделе 4 описаны проводимые с алгоритмом эксперименты и приведены их результаты. Раздел 5 содержит краткий обзор предшествующих работ. В разделе 6 подводятся краткие итоги и обрисовывается предмет дальнейших исследований.

2. Проблемы параллельной разметки

Для задачи параллельной разметки графов объектов существует ряд серьёзных ограничений.

1. Отсутствие общего решения.

Задача разметки графов в общем случае не решается параллельно. В качестве примера достаточно рассмотреть граф, соответствующий односвязному списку, одной из широко используемых структур данных (Рис. 1).

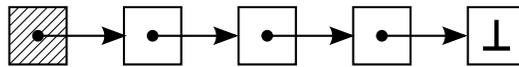


Рис. 1. Контрпример для параллельной разметки графа (заштриховано корневое множество).

В подобных случаях распределение нагрузки между ядрами принципиально невозможно.

2. Мелкозернистый параллелизм

Этой задаче свойственен *мелкозернистый* параллелизм. Размер индивидуальных неделимых подзадач может оказаться мал, вплоть до нескольких инструкций процессора. Поэтому для эффективного параллельного решения задачи необходимо обеспечить балансировку нагрузки путём взаимодействия между потоками с крайне низкими издержками на синхронизацию. Использование стандартных методов синхронизации, таких, как атомарные операции процессора и, тем более, блокировки потоков

средствами операционной системы, не подходит.

3. Деградация производительности

Параллельный алгоритм решения задачи необходим не «ради параллелизма», а как средство достижения изначально поставленной цели — ускорения разметки графа объектов. Это значит, что параллельный алгоритм не должен быть заметно медленнее последовательного в худшем случае. Учитывая контрпример, приведённый в пункте 1, накладные расходы на взаимодействие между потоками, связанные с балансировкой нагрузки, должны быть минимальными.

4. Ограниченность дополнительной памяти

Ещё одна проблема относится скорее к задаче сборки мусора в общем и связана с требованием ограниченности памяти для структур данных самого сборщика мусора. Традиционно такой структурой является стек разметки (*mark stack*) [9]. С одной стороны, если использовать один только стек, его размер в общем случае должен быть пропорционален размеру кучи, так как количество памяти, необходимое для разметки графа в худшем случае, пропорционально количеству его вершин¹. С другой сто-

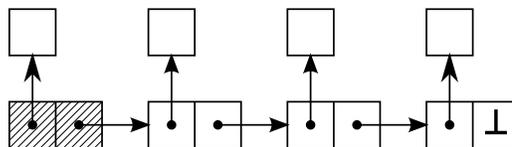


Рис. 2. Граф, требующий $O(n)$ памяти для разметки

роны, резервирование такого объёма памяти заранее противоречит основной задаче эффективного управления памятью. Поэтому сборщики мусора используют вспомогательные структуры данных и алгоритмы, которые применяются в тех случаях, когда стека фиксированного размера не хватает. В случае параллельного алгоритма такие структуры должны быть дополнительно адаптированы к многопоточной работе.

5. Пропускная способность памяти

Масштабируемость любого параллельного алгоритма, основное время работы которого занимает работа с памятью, ограничена пропускной способностью памяти.

В этом можно убедиться, реализовав простую параллельную программу, каждый

¹Поясним это на примере. Рассмотрим алгоритм обхода графа в глубину. Построим граф, состоящий из одной длинной цепи из n вершин, причём каждая вершина ссылается на ещё одну (листовую) вершину так, что при сканировании объектов в стек попадает сначала листовая вершина, а затем — следующая вершина цепи (Рис. 2). Тогда для разметки такого графа стек должен вмещать n элементов.

поток которой выполняет лишь операции чтения больших объёмов памяти.² С увеличением количества работающих потоков при фиксированном объёме работы эта программа работает быстрее (время выполнения становится меньше). Однако, предел этого увеличения меньше, чем количество ядер. Так, например, на четырёхъядерном процессоре Intel Core i5-3470 предельное увеличение достигло примерно 3.19 раз. В аналогичном тесте на запись в память подобный коэффициент оказался ещё ниже: 1.85. Последнее связано с тем, что выполняется не только запись, но и чтение целой линии кэша[6].

Хотя разметка графа объектов состоит не только из операций чтения и записи памяти, она требует интенсивной работы с памятью, поэтому эти константы призваны показать, что предел масштабируемости параллельного алгоритма для неё будет заведомо меньше, чем количество ядер, даже в идеальном случае – при отсутствии зависимостей по данным и издержек синхронизации.

3. Алгоритм

Описанные проблемы накладывают достаточно сильные ограничения на возможные реализации. Именно исходя из этих ограничений мы попробуем построить алгоритм, который будет эффективен для решения поставленной задачи.

Для начала заметим, что синхронизация мелких, связанных с каждым объектом действий недопустима: такой алгоритм не может быть эффективным, поскольку не существует методов синхронизации, накладные расходы которых были бы слабо заметны на фоне работы, связанной с одним объектом.³

Требование 1. *Алгоритм не должен запрещать повторную (и даже одновременную) разметку одного и того же объекта несколькими потоками, как и регистрацию его в структурах данных сборщика.*

Поэтому структурой данных для разметки графа был выбран набор стеков: раз синхронизация на уровне каждого объекта недопустима, у каждого потока исполнения должен быть свой стек.

²Описание этой программы выходит за рамки данной работы. Отметим лишь, что программа учитывает процессорный кэш, предзагрузку памяти (*memory prefetching*) и TLB[6].

³Мы исходим из того, что большинство объектов содержит небольшое количество полей ссылочного типа, сканируемых сборщиком мусора

3.1. Наивный алгоритм

Так мы пришли к простому, «наивному» параллельному алгоритму разметки графа: корневое множество равномерно распределяется между потоками, и затем каждый из них размечает граф, начиная от собственной части корневого множества и используя собственный стек.

Нетрудно, однако, привести пример, на котором такой метод будет неэффективен: размеры подграфов, достижимых из равномошных частей корневого множества, могут быть различными, и тогда, в худшем случае, вся работа достанется одному потоку. Поэтому необходимо распределение нагрузки в ходе разметки графа, т.е. применение какой-либо техники балансировки нагрузки (*load balancing*).

3.2. Алгоритм с балансировкой

Как упоминалось ранее, в худшем случае параллельный алгоритм разметки графа не должен работать заметно медленнее, чем последовательный. Напомним также, что работа может быть распределена по объектам неравномерно.

Требование 2. *Распределением нагрузки должны заниматься не те потоки, у которых есть работа, а те, у которых её нет.*

Предположим обратное, т.е. распределением нагрузки занимаются сами работающие потоки. Неделимым элементом работы будем считать те объекты, которые уже размечены (и помещены в стек), но ещё не просканированы для обнаружения исходящих ссылок. В соответствии с нотацией Дейкстры[9], назовём такие объекты *серыми*. Определение объёма работы, связанной с одним объектом, фактически требует выполнения этой работы, поэтому на этапе распределения нагрузки оценить её невозможно. В худшем случае объекты на стеке вообще не ссылаются на другие объекты, и тогда передача такого объекта другому потоку займёт больше времени, чем просто сканирование. Как следствие, такой алгоритм в худшем случае будет работать заметно медленнее, чем тривиальный однопоточный.

Поэтому в качестве подхода к балансировке нагрузки была выбрана техника, называемая «кража работы» (*work-stealing*)[4, 8].

3.2.1. Техника «кража работы»

Поскольку в нашем алгоритме работа — это сканирование серых объектов со стека разметки, кража работы — это обработка серых объектов с чужого стека.

Для дальнейшего изложения обозначим поток с непустым стеком, выполняющий работу, потоком O (owner), а другой, который пытается украсть у него работу — потоком T (thief).

При отсутствии синхронизации с потоком O , у нас нет никакой возможности явно удалять «украденные» потоком T объекты со стека O . Действительно, между моментом, когда поток T читает элемент стека потока O , и моментом, когда он этот элемент очищает или помечает как «украденный», поток O может переиспользовать этот же элемент стека, помещая туда ссылку на другой серый объект. В этом случае помещённая ссылка будет стёрта, и объект не будет просканирован, т.е. возникнет рассинхронизация данных (*data race*).

Требование 3. *Кража должна состоять лишь в сканировании части объектов с чужого стека. При этом ссылки на эти объекты не должны стираться со стека.*

Таким образом, ссылки на украденные потоком T объекты не стираются, и поток O будет повторно сканировать их. Мы полагаем, что это не должно быть большой проблемой: повторное (но не одновременное) сканирование должно проходить быстрее, так как все объекты, на которые сканируемый объект ссылается, уже помечены при предыдущем сканировании, а значит, повторное сканирование просто ограничится проверкой пометок на объектах-потомках.⁴

При краже поток T сканирует объекты со стека потока O , складывая результаты сканирования (ссылки на объекты-потомки) на свой стек. Можно было бы обойтись простым копированием ссылок со стека O на стек T , но из-за неизбежности повторного сканирования обоими потоками это лишь привнесло бы дополнительные издержки.

Прежде чем перейти к дальнейшему описанию, необходимо сделать небольшое отступление. Нельзя говорить о параллельных алгоритмах, не упомянув синхронизацию кэшей исполняющих устройств (ядер). В рамках данной работы нам будет достаточно знать, что запись в память, копия которой есть в кэше другого ядра, требует синхронизации между кэшами и поэтому вызывает накладные расходы[6]. Отсюда можно вывести ещё два

⁴В качестве альтернативы безусловному повторному сканированию можно явно пометать *чёрные* в нотации Дейкстры[9] (отсканированные) объекты.

важных принципа:

1. Нужно избегать ситуаций, когда разные потоки пытаются размечать одни и те же объекты.
2. Нельзя допускать, чтобы поток Т часто читал память, в окрестность⁵ которой часто пишет поток О. Это касается, например, верхних элементов стека и указателя на вершину стека.

Для соблюдения этих принципов, введём следующие требования.

Требование 4. *Красть ссылки нужно начиная со дна стека.*

Поток О дойдёт до них в последнюю очередь, поэтому есть больше шансов избежать одновременной работы двух потоков над одними и теми же объектами.

Требование 5. *Нужно избегать кражи ссылок, близких к вершине стека.*

Они будут размечаться потоком О в первую очередь, и поток О часто пишет в память около вершины стека.

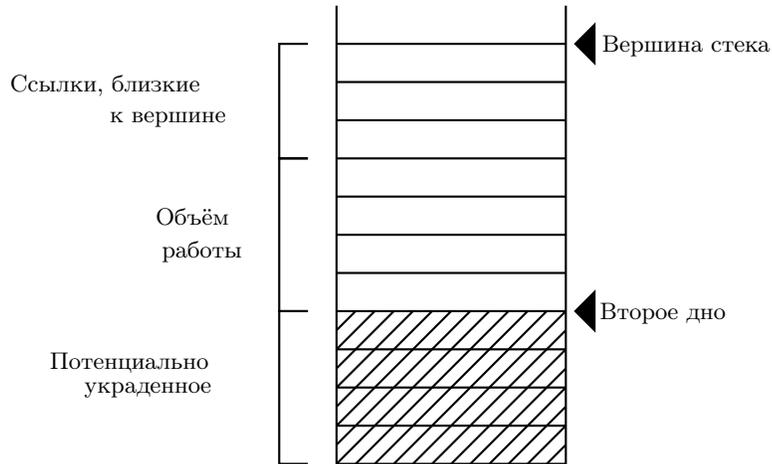
Требование 6. *Нельзя допускать одновременную кражу работы одного потока несколькими другими*

Иными словами, для каждого потока О в каждый момент времени существует только один поток Т. Это ограничение упрощает весь алгоритм в целом и задачи синхронизации в частности.

Требование 7. *Необходимо обеспечить, чтобы разные потоки не крали одни и те же объекты.*

Для этого, помимо указателя на вершину стека, введём ещё один указатель (назовём его «вторым дном» стека). Его семантика будет такова: «Выше второго дна ещё никто ничего не крал» (Рис. 3).

⁵Имеется ввиду та же линия кэша



Ссылки ниже второго дна могли быть уже украдены. Между вторым дном и вершиной стека гарантированно ещё ничего не украдено, и поэтому ссылки из этого промежутка (кроме ссылок, близких к вершине стека) являются целью для кражи.

Рис. 3. Стек с двойным дном

Изменения этого указателя будут двух видов:

1. **Повышение.** Поток T устанавливает его после кражи в положение выше последней украденной ссылки. Заметим, что мы не можем гарантировать, что ниже этого указателя всё украдено, так как стек при этом используется ещё и потоком O .
2. **Понижение.** Поток O не даёт этому указателю превысить указатель на вершину стека. В случае, когда вершина стека опускается ниже, второе дно устанавливается в то же положение, т.е. равным вершине стека. Этим мы обеспечиваем то, что новые объекты на стеке смогут попадать выше второго дна.

Если потоки крадут работу начиная со второго дна вверх по стеку, то они гарантированно избегают кражи тех объектов, которые уже были украдены ранее.

Требование 8. *Количество украденных за один раз объектов нужно ограничить.*

Если поток T крадёт за один раз большое количество объектов, он может получить больше работы, чем требовалось, оставив поток O без работы. С другой стороны, нельзя красть слишком мало, иначе накладные расходы, связанные с поиском работы и началом кражи, будут заметны. В прототипе за один раз поток T сканировал не больше 64-х объектов.

3.2.2. Поиск работы

Когда у потока T заканчивается работа, он ищет, у кого её можно украсть. Для этого он проверяет каждый поток на предмет того, можно ли украсть работу у него. Если такого

потока не находится, то поток Т выполняет короткое ожидание⁶, а затем повторяет процесс поиска. В этой простой схеме существует две проблемы. Первая связана с определением момента остановки работы алгоритма разметки, и она обсуждается в разделе 3.2.3. Вторая связана с неэффективностью определения того, есть ли у другого потока работа, которую можно украсть.

Согласно 3.2.1, можно сказать, что потоку Т есть что воровать у потока О в том случае, если между вершиной стека и вторым дном есть большой промежуток (объём работы на рис. 3). Проблема в том, что проверять это напрямую неэффективно: оба указателя активно изменяются потоком О в процессе работы, и поэтому частое их чтение другими потоками⁷ приводит к существенному замедлению работы потока О.

Вместо чтения указателей поток Т будет читать специальный флаг *hasWork*, выставяемый потоком О (*O.hasWork*), означающий, что у потока О есть работа, которую можно украсть. Чтобы не оказывать большого влияния на производительность потока О, выставление *hasWork* будет производиться раз в некоторое количество операций со стеком (в прототипе используется 256). Кроме того, *hasWork* сбрасывается потоком Т после кражи в том случае, если он украл всю имеющуюся работу. Этот флаг расположен на отдельной линии кэша, чтобы он не оказался в той же линии кэша, что и другие часто используемые в работе данные, для того, чтобы избежать проблемы *false-sharing*[3]. Псевдокод схемы работы потока О приведён на рис. 4.

цикл

если свой стек пуст **то**

выйти из цикла

конец

извлечь объект со своего стека и просканировать его

если второе дно > указатель на вершину стека **то**

второе дно ← указатель на вершину стека

конец

выставить *hasWork* в зависимости от вершины стека и второго дна

конец

Рис. 4. Схема работы потока О

Наконец, в соответствии с требованием б, у каждого потока в каждый момент времени красть работу может только один поток. Для этого каждый поток имеет мьютекс (*O.mutex*), который исключает одновременные кражи у потока О. Этот мьютекс захва-

⁶В целях производительности такое ожидание должно быть реализовано без блокировки потока средствами операционной системы

⁷А их чтение будет частым в том случае, когда у большого числа потоков нет работы

тывается и освобождается потоком T и может быть тривиальным образом реализован через атомарный флаг. Псевдокод схемы кражи работы потоком T у потока O приведён на рис. 5.

```

если  $O.hasWork$  то
  попыбовать захватить  $O.mutex$ 
если удалось захватить то
  украсть работу
  освободить  $O.mutex$ 
конец
конец

```

Рис. 5. Схема работы потока T

3.2.3. Проблема останова

Каждый работающий поток должен завершиться вовремя, то есть

1. Поток не должен завершиться раньше, чем просканированы все объекты
2. После того, как все объекты просканированы, все потоки должны завершиться.

Для решения этой проблемы введено два возможных состояния потока:

- **Занят.**

Поток работает. К этому относится как кража работы, так и сканирование объектов с собственного стека.

- **Свободен.**

Поток ищет работу.

Переход из первого состояния во второе происходит после того, как поток закончил обработку собственного стека, из второго в первое — в момент, когда он нашёл, у кого можно украсть работу, перед непосредственной кражей.

Как только бездействующий поток видит, что в какой-то момент времени все остальные потоки также бездействуют, он должен завершиться. Для того, чтобы сделать такую проверку потоково-безопасной, вводится *counter* – атомарный счётчик⁸ числа потоков, находящихся в состоянии "занят". Как только очередной поток выходит из состояния "занят", он уменьшает счётчик и проверяет, не стал ли он равен нулю. В последнем случае выставляется *exit* – глобальный флаг завершения. Каждый поток периодически (между попытками кражи) проверяет значение этого флага.

⁸Для проблемы останова было найдено решение, не использующее атомарные операции процессора, но оно не принесло видимых улучшений в производительности и поэтому, будучи более сложным, в работе не описывается

На рис. 6 приведён псевдокод общей схемы алгоритма. Изначально *counter* равен количеству используемых потоков, а *exit* равен *false*. Вложенный цикл — это цикл, в котором поток находится, пока ищет работу или ждёт завершения.

цикл

обработать собственный стек согласно рис. 4

$val \leftarrow$ атомарно уменьшить *counter* на 1 и взять новое значение

если $val = 0$ **то**

$exit \leftarrow true$

конец

цикл

если *exit* **то**

завершиться

конец

если поток может украсть работу согласно рис. 5 **то**

атомарно увеличить *counter* на 1

украсть работу

выйти из цикла

конец

ожидание

конец

конец

Рис. 6. Общая схема алгоритма (тело рабочего потока)

Теорема 1 (о корректности). После завершения алгоритма все достижимые объекты помечены.

Доказательство. Пусть алгоритм завершился. Тогда флаг *exit* был выставлен, следовательно, в какой-то момент времени *counter* оказался равен нулю после очередного уменьшения. Это значит, что на этот момент все стеки пусты. Рассмотрим совокупность стеков как очередь серых объектов в алгоритме разметки графа. Поскольку каждый объект удаляется из этой очереди только после того, как был просканирован, и в данный момент очередь пуста, то все достижимые объекты уже помечены. □

Теорема 2 (о завершимости). Алгоритм завершается.

Доказательство. Пусть *counter* оказался равен нулю после очередного уменьшения. Тогда выставляется флаг *exit*. Поскольку все стеки на этот момент пусты, и ни один поток не находится в процессе кражи, все дальнейшие операции в псевдокоде будут выполняться

быстро (поскольку станут тривиальными), и поэтому через короткое время каждый поток прочитает выставленный флаг *exit* и завершит работу.

Таким образом, все потоки завершат работу вскоре после того, как счётчик *counter* обнулится. Покажем, что рано или поздно это произойдёт. Действительно, каждый объект может попасть на каждый стек не более одного раза (в момент маркировки). При этом каждый элемент каждого стека может быть просканирован не больше двух раз (один раз – хозяином стека, второй – вором). Таким образом, каждый объект может быть просканирован не более $2p$ раз, где p – это число потоков. Пусть n – это число достижимых объектов, тогда общее число операций сканирования ограничено сверху $2pn$. Пока счётчик не равен нулю, найдётся поток, который занят сканированием объектов. Таким образом, общее число выполненных операций сканирования монотонно возрастает. А значит, раз ограничение не может быть превышено, рано или поздно счётчик обнулится, и тогда все потоки будут завершены. \square

Но как скоро после фактического выполнения всей работы *counter* обнулится? Мы не можем гарантировать, что это случится *сразу после того*, как все достижимые объекты будут помечены и просканированы: раз ссылки не удаляются при краже, потоки ещё просканируют и снимут все объекты со своего стека, не кладя ничего нового.

3.3. Анализ худших случаев

Несмотря на все попытки избежать проблем, связанных с неравномерностью распределения работы по стекам, предложенный алгоритм им подвержен, так что возможно построить очевидные контрпримеры, основанные на этой неравномерности.

Один из таких примеров изображён на рисунке 2. При разметке такого графа объектов поток O будет размечать список: каждый раз, снимая со стека очередной элемент списка, он кладёт на стек листовую вершину и следующий элемент списка, который будет снят с вершины стека на следующей итерации. Таким образом, на стеке накапливаются листовые объекты. Любой поток T , который будет красть работу у потока O , будет сканировать листовые вершины, не совершая никакой полезной работы. При этом каждый из этих объектов будет повторно просканирован потоком O . Учитывая ограниченность пропускной способности памяти, выполнение лишних операций с памятью и взаимодействие между потоками приводит к деградации производительности (см. раздел 4.3).

Подойдём к этому вопросу более формально. Отношение времени работы классическо-

го однопоточного алгоритма ко времени работы параллельного алгоритма назовём *ускорением* этого параллельного алгоритма. Величина ускорения, меньшая 1, означает, что наблюдается не ускорение, а замедление (деградация производительности). Тогда контр-примером будем считать входные данные, на которых ускорение минимально.

Пусть однопоточный алгоритм работает за время $c_1 \cdot N + c_2 \cdot M$, где N – это число объектов, а M – число ссылочных полей в этих объектах.

Рассмотрим алгоритм с балансировкой. В общем случае анализ времени его работы достаточно затруднён. Будем искать худшие случаи среди тех, где есть поток O , который всё время занят обработкой своего стека, т.е. ничего не крадёт. Тогда время работы всего алгоритма равно времени работы потока O . Пусть поток O обрабатывает n объектов, у которых есть m ссылок, и за это время у него украли (т.е. просканировали с его стека) s объектов. Во-первых, часть времени его работы занимает обработка объектов, т.е. $c_1 \cdot n + c_2 \cdot m$, как у однопоточного алгоритма. Но у алгоритма также есть и накладные расходы. Во-первых, это расходы на проверки, нужно ли выставлять флаг наличия работы (см. раздел 3.2.2). Поскольку они привязаны к операциям со стеком, мы будем считать, что они пропорциональны количеству объектов, т.е. n . Во-вторых, непосредственная кража объектов также вызывает накладные расходы, причём, не только у вора, но и у хозяина стека. Будем считать, что они пропорциональны числу украденных объектов. Таким образом, время работы потока O составит $c_1 \cdot n + c_2 \cdot m + c_3 \cdot n + c_4 \cdot s$.

Мы ищем худшие случаи, то есть, пытаемся минимизировать ускорение. Иными словами, мы пытаемся максимизировать отношение:

$$\max \leftarrow \frac{c_1 \cdot n + c_2 \cdot m + c_3 \cdot n + c_4 \cdot s}{c_1 \cdot N + c_2 \cdot M} \quad (3.1)$$

При фиксированных N и M это отношение тем больше, чем больше n , m и s . Значит, максимум следует искать среди тех случаев, где поток O обрабатывает весь граф, т.е.

$$\max \leftarrow \frac{c_1 \cdot N + c_2 \cdot M + c_3 \cdot N + c_4 \cdot s}{c_1 \cdot N + c_2 \cdot M} = 1 + \frac{c_3 \cdot N + c_4 \cdot s}{c_1 \cdot N + c_2 \cdot M} \quad (3.2)$$

Зафиксируем N . Тогда чем меньше M и чем больше s , тем больше (3.2). Но, с другой стороны, $s \leq N$ и $M \geq N - 1$. Таким образом, если мы представим граф объектов, в котором M и s близки к N , его можно будет считать худшим случаем.

Вернёмся к рисунку 2. Этот граф соответствует тому требованию, что все объекты

фактически обрабатываются одним потоком. При этом $s = N/2$, а $M = N$. Можно ли составить граф, где эти показатели были бы ещё хуже?

Пусть в графе на рисунке 2 каждый элемент списка непосредственно ссылается не на один листовой объект, а на k . Здесь по-прежнему $M = N$. При этом каждый листовой объект будет украден, поэтому $s = N \cdot \frac{k}{k+1}$. Таким образом, увеличивая k , можно составить сколь угодно близкий к худшему случаю граф.

4. Экспериментальные результаты

4.1. Методология эксперимента

Описанный алгоритм был реализован в рамках исследовательской виртуальной Java машины Excelsior RVM[10], которая поддерживает полный стандарт платформы Java 8.

Измерения проводились следующим образом. В качестве длительности разметки графа объектов, на основании которой подсчитывается ускорение (см. раздел 3.3), использовалась суммарная длительность фаз разметки по всем вызовам сборщика мусора во всех запусках теста. Каждый из тестов запускался по 3-4 раза. Поскольку разброс результатов между различными запусками тестов был не слишком велик, такого числа измерений хватает для оценки.

Для измерений виртуальная машина была сконфигурирована так, чтобы выполнялись только полные циклы сборки мусора, во время которых размечается весь граф объектов. Это было сделано для увеличения нагрузки.

4.2. Тестовый набор

Измерения проводились на тестах двух категорий. Первая состоит из разметки синтетических графов, таких как односвязные и двусвязные списки, бинарные деревья и леса, и так далее. Несмотря на то, что алгоритм предназначен для разметки графов объектов реальных приложений, а не синтетических тестов, использование таких тестов помогает выявить патологические случаи и улучшить множество аспектов алгоритма. Это позволит в общем случае получить прирост производительности и на реальных примерах.

В качестве синтетических графов использованы двусвязный список (DList), наборы односвязных списков (SList, SListMultiple), односвязный список с листовыми вершинами (SListML) (см. раздел 3.3), односвязный список с присоединёнными к элементам небольшими подграфами (SListD), бинарные леса (Tree2, Tree2Multiple), тернарное дерево (Tree3)

и «список с вкраплениями алмазов» (DiamondList), который строится таким образом: вводится последовательность основных вершин, и каждые две соседние соединяются поочередно одним либо двумя односвязными списками (Рис. 7).

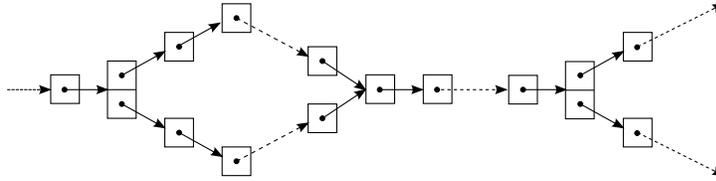


Рис. 7. DiamondList

Число вершин и дуг, а также мощность корневого множества для графов синтетических тестов приведены в таблице 1. Тесты с «маломощным» корневым множеством предназначены для оценки качества балансирования нагрузки.

Тест	Кол-во вершин	Кол-во дуг	Корневое множество
DiamondList	15250021	15500020	1
DList	20000001	40000001	1
SList	19000002	19000000	2
SListD	16000032	32000063	1
SListL	2000002	2000001	1
SListML	1050021	1050020	1
SListMultiple	20000040	20000000	40
Tree2	67108863	134217726	1
Tree2Multiple	20971480	41942960	40
Tree3	21523360	64570080	1

Таблица 1

Характеристики синтетических тестов

Вторая категория — это разметка графов объектов реальных приложений. В список приложений входят тесты производительности DaCapo.ps[2] (рендеринг PostScript-документа), SPECjbb2000[12] (трёхуровневая клиент-серверная система) и SPECjvm2008.derby[11] (нагрузочный тест для СУБД Apache Derby).

В синтетических тестах сборка мусора вызывалась вручную по 10 раз. В тестах с графами объектов реальных приложений вызовы сборки мусора осуществлялись так же, как и при обычной работе этих приложений.

4.3. Результаты и обсуждение

Измерения проводились на компьютере, оснащённом четырёхъядерным процессором Intel Core i5-3470 с частотой ядер 3200 MHz с 8 GB оперативной памяти под управлением

64-битной операционной системы Windows 7. Процессор содержит 64 Кб кэша первого уровня и 256 Кб кэша второго уровня для каждого ядра и 6 Мб общего кэша третьего уровня.

В таблице 2 приведены результаты измерений. Первая колонка содержит название теста, вторая и третья — ускорения (см. разд. 3.3) наивного параллельного алгоритма и алгоритма с балансировкой по отношению к однопоточному, соответственно.

Тест	naïve	stealing
DList	1.02	0.91
DiamondList	1.00	1.23
SList	1.95	1.80
SListD	1.00	1.79
SListML	0.99	0.88
SListMultiple	2.64	2.65
Tree2	1.03	2.54
Tree2Multiple	3.60	3.50
Tree3	1.06	3.40
SPECjbb2000	2.28	2.57
SPECjvm2008.derby	1.28	2.46
DaCapo.ps	1.00	1.76

Таблица 2

Ускорение разметки

Наличие небольших ускорений в тех тестах, где их вообще быть не должно⁹, объясняется тем, что помимо размечаемого синтетического графа в куче есть также и другие объекты в небольшом количестве, и их удаётся размечать параллельно. На тесте SListML наблюдается ожидаемое замедление. Напомним, что тест SListML является контрпримером к алгоритму. Он основан на том, что динамическое распределение работы на таком графе не приносит никакой выгоды (см. раздел 3.3). На тесте DList работу распределить нельзя, поэтому в качестве причины такой деградации можно подозревать только дефекты в реализации протокола опроса работающего потока ворующими (раздел 3.2.2) или в алгоритме останова (раздел 3.2.3). Исследование причин, вызывающих такое замедление, оставлено на будущее.¹⁰ Тесты Tree2 и Tree3 показывают хорошую работу механизма балансировки нагрузки: наивный алгоритм на них не даёт преимуществ, тогда как алгоритм с балансировкой показывает значительное увеличение производительности. На тестах SListD и DiamondList, которые можно считать хорошими случаями для алгоритма,

⁹Например, для наивного алгоритма это тесты с одноэлементным корневым множеством

¹⁰Есть предположение, что одной из основных причин замедления здесь является *false sharing*[3]

также можно видеть преимущество балансирующего алгоритма над наивным. На тестах SListMultiple и Tree2Multiple видно, что балансировка нагрузки не всегда вносит значительные деградации в наивный параллельный алгоритм: на каждом из этих тестов оба ускорения сравнимы. На рис. 8 приведены графики ускорений обоих алгоритмов. Наконец, тесты SPECjbb2000, SPECjvm2008.derby и DaCapo.ps показывают, что балансировка нагрузки даёт существенный выигрыш на графах реальных приложений. На рис. 9, 10 и 11 приведено сравнение роста ускорений алгоритмов на этих тестах.

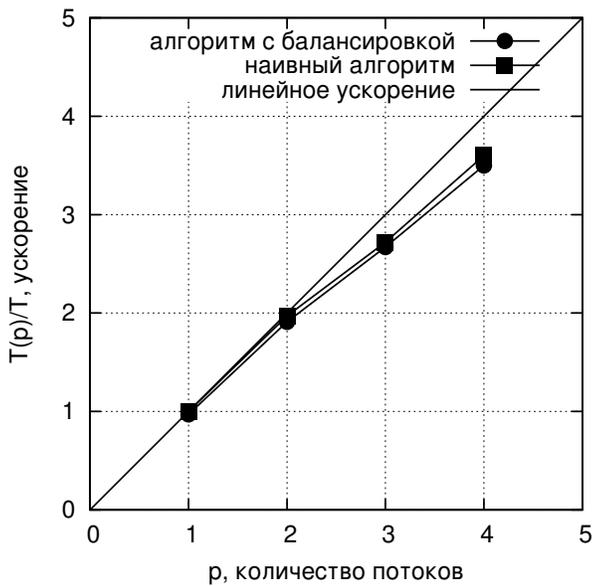


Рис. 8. Ускорения разметки на тесте Tree2Multiple

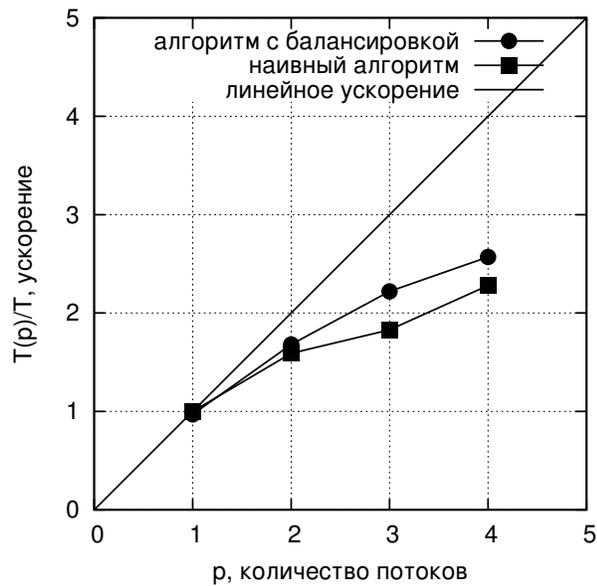


Рис. 9. Ускорения разметки на тесте SPECjbb2000

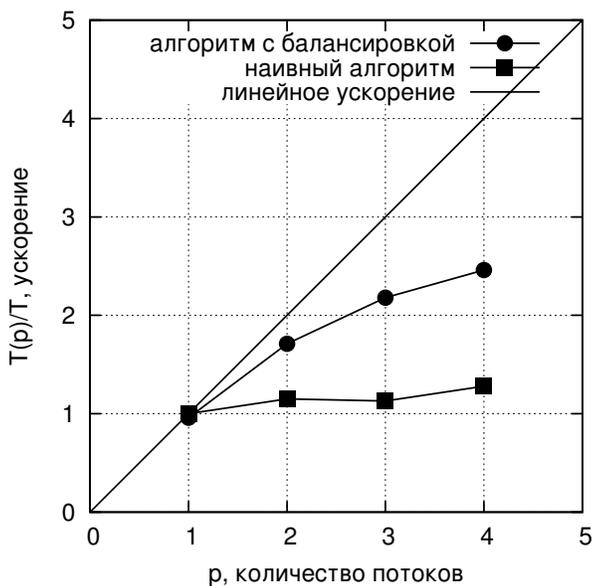


Рис. 10. Ускорения разметки на SPECjvm2008.derby

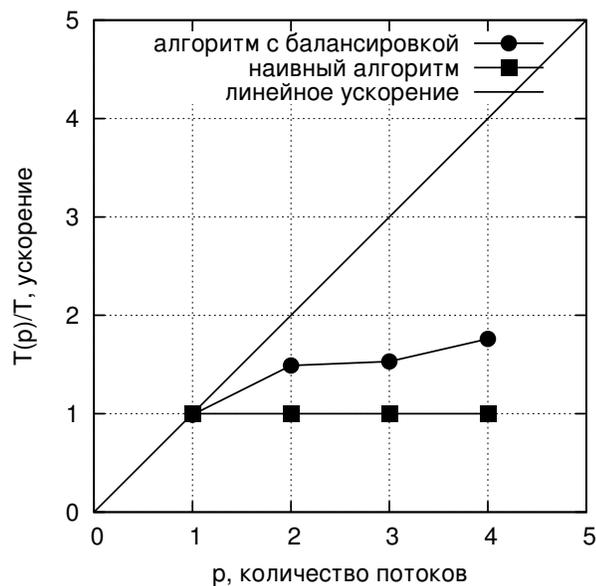


Рис. 11. Ускорения разметки на тесте DaCapo.ps

Предложенный алгоритм показывает здесь хороший результат, что подтверждает адек-

ватность положенных в его основу идей.

5. Обзор предшествующих работ

В данном разделе приведён критический обзор работ, предлагающих различные подходы к параллельной разметке графа объектов.

В работе [7] описаны параллельные версии разных алгоритмов, используемых классическими сборщиками мусора. Авторы отмечают необходимость динамической балансировки нагрузки для разметки графа объектов и так же, как и мы, выбирают технику кражи работы. В работе предлагается снабдить каждый поток отдельной структурой данных для разметки. В качестве таковой выбирается не стек, а специальная очередь с тремя операциями:

1. Добавление в начало потоком O ; не требует синхронизации
2. Извлечение из начала потоком O ; синхронизация используется в том случае, когда извлекается последний элемент
3. Извлечение с конца произвольным потоком T ; синхронизация используется всегда

В работе потока O используются первые две операции, поэтому связанные с синхронизацией накладные расходы для потока O незаметны.

Однако, операция извлечения элемента из начала такой очереди гораздо сложнее, чем операция извлечения элемента с вершины обычного стека, что должно отрицательно сказываться на производительности в худшем случае. В статье не приведены измерения работы алгоритма на синтетических графах объектов (например, на односвязном списке), поэтому оценить степень деградации не представляется возможным. В целом подход выглядит удачным по результатам измерений на реальных приложениях, однако размеры некоторых из графов объектов этих приложений недостаточно велики.

В статье [14] представлена иная техника динамической балансировки. Авторы описывают специальную структуру данных — очередь для передачи задач от одного потока к другому, которая не требует синхронизации. Основная идея такой очереди в том, что в ней разрешены только две операции: один фиксированный поток может извлекать элементы из начала очереди, а другой — добавлять в конец. Авторы предлагают использовать очередь малой длины, но при этом не адресуют проблему замедления, вызванную синхронизацией кэшей процессоров. При этом результаты измерений на синтетических тестах, которые показали бы масштаб проблемы, в работе не приведены, не анализируются контрпримеры,

а общее количество тестов мало. Кроме того, в результатах встречаются маловероятные, с нашей точки зрения, данные. Так, например, согласно графикам, добавление атомарной операции для разметки каждого объекта влияет на производительность лишь незначительно, что разительно расходится с нашим опытом. Таким образом, данная работа после изучения оставляет ряд вопросов, и для сравнения результатов с другими работами необходимо реализовать предлагаемый алгоритм и измерить его характеристики независимо.

6. Заключение

В настоящей работе описаны основные проблемы, возникающие при параллельной разметке графов объектов в трассирующих сборщиках мусора, и приведён алгоритм, который призван эти проблемы решать. Алгоритм был реализован в управляемой среде Excelsior JVM[10], а проведенные измерения показали обоснованность выбранных решений. Так, коэффициент ускорения на 4-х ядерной машине при тестировании реальных приложений варьируется от 1.76 до 2.57.

Несмотря на то, что прототип продемонстрировал хорошие результаты на графах объектов реальных приложений, решены ещё не все проблемы с производительностью на некоторых синтетических тестах. Также не проведено непосредственное сравнение производительности описанного алгоритма с алгоритмами из других работ [7, 14], что требует их реализации в той же управляемой среде.

Некоторые допущения при разработке алгоритма основаны на опыте или результатах экспериментов. Они могут быть слишком категоричными и затрагивать только частные случаи, поэтому необходимо исследовать и другие возможные подходы к параллельной разметке.

Наконец, задача параллельной разметки может и должна рассматриваться в контексте всей задачи сборки мусора. Например, фаза построения корневого множества для разметки графа, которое меняется в ходе работы программы, также является вычислительной задачей и может быть выполнена параллельно. Кроме того, стратегия распределения корневого множества между работающими потоками обладает рядом степеней свободы, и выбор правильной схемы может быть обусловлен анализом корневого множества с опорой на семантику среды исполнения и программы. Перечисленные задачи составляют предмет дальнейших исследований.

Список литературы

1. Appel A. W. Simple generational garbage collection and fast allocation // Software Practice & Experience. 1989. Vol. 19, 2. P. 171–183.
2. Blackburn S. M., Garner R., Hoffmann C. et al. The DaCapo benchmarks: java benchmarking development and analysis // Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications. New York, NY, USA: ACM, 2006. P. 169–190.
3. Bolosky W. J., Scott M. L. False Sharing and its Effect on Shared Memory Performance // Proceedings of the USENIX SEDMS IV Conference. 1993.
4. Burton F. W., Sleep M. R. Executing functional programs on a virtual tree of processors // Proceedings of the 1981 conference on Functional programming languages and computer architecture. 1981.
5. Click C., Tene G., Wolf M. The Pauseless GC Algorithm // Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments. VEE '05. New York, NY, USA: ACM, 2005. P. 46–56. URL: <http://doi.acm.org/10.1145/1064979.1064988>.
6. Drepper U. What Every Programmer Should Know About Memory. <http://people.redhat.com/drepper/cpumemory.pdf>.
7. Flood C. H., Detlefs D., Shavit N., Zhang X. Parallel garbage collection for shared memory multiprocessors // Proceedings of the 2001 Symposium on Java™ Virtual Machine Research and Technology Symposium - Volume 1. JVM'01. Berkeley, CA, USA: USENIX Association, 2001. P. 21–21.
8. Halstead R. H. Implementation of multilisp: Lisp on a multiprocessor // Proceedings of the 1984 ACM Symposium on LISP and functional programming. 1984.
9. Jones R., Lins R. Garbage collection: algorithms for automatic dynamic memory management. New York, NY, USA: John Wiley & Sons, Inc., 1996.
10. Mikheev V., Lipsky N., Gurchenkov D. et al. Overview of excelsior JET, a high performance alternative to java virtual machines // Proceedings of the 3rd international workshop on Software and performance. New York, NY, USA: ACM, 2002. P. 104–113.
11. SPEC releases free SPECjvm2008 benchmark. Press release. URL: <http://www.spec.org/jvm2008/press/release.html>.
12. SPECjbb2000, A Java Business Benchmark. White paper. URL: <http://www.spec.org/>

jbb2000/docs/whitepaper.html.

13. Ungar D. Generation Scavenging: A non-disruptive high performance storage reclamation algorithm // Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments. New York, NY, USA: ACM, 1984. P. 157–167.
14. Wu M., Li X.-F. Task-pushing: a Scalable Parallel GC Marking Algorithm without Synchronization Operations // IEEE International Parallel and Distributed Processing Symposium. 2007.