МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ

И.А. Бессмертный

ИСКУССТВЕННЫЙ ИНТЕЛЛЕКТ

Учебное пособие

Санкт-Петербург

2010

И.А.Бессмертный. Искусственный интеллект – СПб: СПбГУ ИТМО, 2010. – 132 с.

Настоящее учебное пособие разработано в рамках дисциплины «Искусственный интеллект», преподаваемой на кафедре вычислительной техники СПбГУИТМО и включает в себя основы программирования на языке Prolog, решение задач методом поиска, вероятностные методы, основы нейронных сетей, а также принципы представления знаний с помощью семантических сетей. Каждый из разделов учебного пособия обеспечен практическими и лабораторными работами. В приложениях содержатся краткие описания среды SWI-Prolog, программы нейросетевого моделирования NeuroGenetic Optimizer и программы визуализации знаний Semantic.

Для студентов специальностей 23010111, 23010104, 23010011, 23010020, 23010031, 023010032

Рекомендовано к печати ученым советом факультета КТиУ 19.01.2010, протокол №6.

В 2009 году Университет стал победителем многоэтапного конкурса, в результате которого определены 12 ведущих университетов России, которым присвоена категория «Национальный исследовательский университет». Министерством образования и науки Российской Федерации была утверждена Программа развития государственного образовательного учреждения высшего профессионального образования «Санкт-Петербургский государственный университет информационных технологий, механики и оптики» на 2009–2018 годы.

ОГЛАВЛЕНИЕ

Введение

История искусственного интеллекта (ИИ) начинается задолго до нашей эры. Аристотель был первым, кто попытался определить законы «правильного мышления» или процессы неопровержимых рассуждений. Попытки создания механических счетных устройств в средние века сильно впечатляли современников. Наиболее известна машина Паскалина, построенная в 1642 г. Блезом Паскалем. Паскаль писал, что «арифметическая машина производит эффект, который кажется более близким к мышлению по сравнению с любыми действиями животных».

Возможности же практической реализации ИИ появились с момента создания электронных вычислительных машин. В это время развернулась философская дискуссия на тему «Может ли машина мыслить?». Итогом этой дискуссии стал тест, предложенный Аланом Тьюрингом в 50-е гг. XX века [1]. Тест заключается в следующем: Имеются два телетайпа (в то время других терминальных устройств не было, сейчас бы предложили ICQ). Один из телетайпов подключен к машине, другой — к аппарату, за которым сидит человек. Несколько экспертов поочередно ведут диалог на каждом из телетайпов. Если большинство экспертов не смогут в течение пяти минут распознать в одном из собеседников машину, то тест Тьюринга считается пройденным успешно.

Тест Тьюринга сыграл определенную роль в развитии искусственного интеллекта, в том числе и критика самого теста. Здесь можно провести аналогию с авиацией. Хорошими летательными аппаратами, по логике теста Тьюринга, должны считаться такие, которые неотличимы от птиц до такой степени, что даже птицы принимают их за своих. Развитие авиации началось тогда, когда конструкторы перестали копировать птиц, а занялись аэродинамикой, материаловедением и теорией прочности. Робототехника стала индустрией после того, как перестала копировать анатомию человека.

Аналогично, субъекты искусственного интеллекта получили право на жизнь после того, как прекратились попытки построить системы ИИ, думающие и действующие подобно людям, а начали строить системы, **действующие и думающие рационально**, т.е. достигающие наилучшего результата.

Последние достижения в области ИИ можно представить следующими коммерческими проектами [1]:

- **Автономное планирование и составление расписаний**. Программа Remote Agent, разработанная в NASA, используется для комплексного управления работой космических аппаратов, удаленных далеко за пределы околоземной орбиты, в т.ч. диагностики и устранения неисправностей по мере их возникновения.

- **Ведение игр**. Программа Deep Blue компании IBM стала первой программой, которой удалось победить чемпиона мира в шахматном матче.
- **Автономное управление**. Система компьютерного зрения Alvinn была обучена вождению автомобиля, придерживаясь полосы движения. На протяжении 2850 миль система обеспечивала управлении автомобилем в течение 98% времени.
- **Диагностика.** Медицинские диагностические программы сумели достигнуть уровня опытного врача в нескольких областях медицины.
- **Планирование снабжения**. Во время кризиса в Персидском заливе в 1991г. В армии США была развернута система DART (Dynamic Analysis and Re-planning), которая обеспечивала автоматизированное планирование поставок и составление графика перевозок, охватывая одновременно до 50000 автомобилей, людей и грузы. Разработчики этой системы заявили, что одно это применение окупило их 30-летние инвестиции в искусственный интеллект.

1. Основы программирования на языке Prolog

1.1. Prolog как декларативный язык

Разработка языка Prolog началась в 1970 г. Аланом Кулмероэ и Филиппом Русселом. Они хотели создать язык, который мог бы делать логические заключения на основе заданного текста. Название Prolog является сокращением от "PROgramming in LOGic". Этот язык был разработан в Марселе в 1972г.

Prolog – язык программирования, который основан не на алгоритме, а на логике предикатов. Если программа на алгоритмическом (процедурном) языке является последовательностью инструкций, выполняющихся в заданном порядке, то программа на Прологе содержит только описание задачи, а Пролог-машина выполняет поиск решения, руководствуясь этим описанием. Например, существует логическая задача покрытия шахматной доски ходом коня. На любом алгоритмическом языке решение этой задачи требует построения достаточно сложного алгоритма. На Прологе достаточно описать правила, по которым ходит конь, после чего Пролог самостоятельно отыщет решение. Обратной стороной такой простоты является ресурсоемкость программ. Например, в другой популярной задаче размещения на шахматной доске восьми ферзей, которые не бьют друг друга, полное дерево решений имеет 64^8 вершин. Очевидно, что нахождение решения в таком дереве займет неприемлемо много времени.

Программирование на языке Пролог состоит из следующих этапов:
• объявления некоторых фактов об объектах и отношениях между ними,
• определения некоторых правил об объектах и отношениях между ними;
• формулировки вопросов об объектах и отношениях между ними.

1.2. Понятие предиката

Основным элементом программы на Прологе является предикат. С математической точки зрения предикат – это функция, которая возвращает бинарное значение (истина или ложь). В Прологе предикатом обозначается отношение между объектами, которое также может быть истинным.

Рассмотрим понятие предиката в Прологе на примере звездной семьи Пугачевой – Киркорова, правда, теперь уже бывшей. Вначале запишем отношения типа родитель – ребенок. В синтаксисе Пролога выражение «Борис является родителем Аллы» выглядит следующим образом:

parent(boris, alla).

Здесь *parent* – это имя предиката, а *boris* и *alla* – аргументы. Аргументы *boris* и *alla* являются константами, поэтому записаны строчными буквами. С прописной буквы в Прологе начинаются переменные. Точка означает конец предиката, так же, как и конец предложения на естественном языке. Запишем также родительские отношения для других членов семьи:

parent(bedros, filipp). *parent(kristina, denis).*
parent(edmuntas, kristina). *parent(vladimir, denis).*
parent(alla, kristina).

Теперь дадим понятие «супруг» (spouse):

spouse(filipp, alla).
spouse(vladimir, kristina).

Полученный набор предикатов образует базу знаний о звездной семье. Сравним с тем, как те же данные будут представлены в реляционной базе.

Таблица Родители	
Parent	**Child**
Boris	*Alla*
Bedros	*Filipp*
Alla	*Kristina*
Edmuntas	*Kristina*
Kristina	*Deni*

Таблица Супруги	
S1	**S2**
Alla	*Filipp*
Kristina	*Vladimir*

Как видим, на уровне представления данных сходство налицо. Но на этом оно и кончается. Но на уровне извлечения данных имеет место большое различие. Реляционной базе данных для того, чтобы извлечь знания, требуется создать запрос на выборку данных, например, на языке SQL.

Пусть, мы хотим узнать, кто родители Кристины. Мы должны написать запрос следующего вида:

SELECT *Parent* FROM *Родители* WHERE *Child = "Kristina"*

В Прологе запрос на извлечение знаний описывается такими же предикатами, какими эти знания представляются. Если мы подставим Прологу такой предикат (предикат цели):

parent(alla, kristina).

Эта цель может быть прочитана следующим образом: Является ли Алла родителем Кристины? Сопоставив эту цель с содержимым базы знаний, Пролог установит, что данное утверждение истинно и сообщит об этом.

Вышеприведенный SQL запрос (Кто является родителем Кристины?) в Прологе выглядит следующим образом:

parent(X, kristina).

Здесь *X* является переменной, которой должны быть присвоены искомые значения. Переменная в Прологе является аналогом местоимения или вопросительного слова. Из приведенной выше базы знаний Пролог извлечет два ответа:

X = alla
X = edmuntas

Мы можем сформулировать вопрос следующим образом: Есть ли у Кристины родители?

parent(_, kristina).

Пролог выдаст ответ: *Yes*.

Переменная, начинающаяся со знака подчеркивания называется анонимной переменной и может принимать любые значения (аналог местоимения некто).

Для более сложных запросов в базах данных необходимо создавать представления (view) или создавать вложенные запросы в SQL. В Прологе все гораздо проще. Давайте найдем, чьей внучкой является Кристина. Запрос будет выглядеть следующим образом:

parent(X, kristina), parent(Y, X).

Данная запись означает: Найти *Y*, являющийся родителем *X*, который, в свою очередь, является родителем Кристины. Запятая в прологе идентична союзу "И" или конъюнкции. В ответ на такой запрос Пролог выдаст следующий ответ:

X = alla
Y = edmuntas

Можно выдать следующий запрос:

parent(alla, _).

Теперь анонимная переменная использована в качестве второго аргумента. Этот запрос можно прочитать следующим образом: Есть ли у Аллы дети? Пролог выдаст ответ: *Yes*.

Поскольку предикат – это бинарная функция, которая может возвращать истину либо ложь (высказывание, которое может также быть истинным или ложным), результат работы программы на Прологе – это определение, истинна или ложна цель. Присвоение значение переменным, вывод результатов и т.п. –

это всего лишь побочные результаты.

Таким образом, программа на Прологе состоит из предикатов. Программа на Прологе и база знаний - синонимы. Цель формулируется также в виде предикатов. Выполнение программы на Прологе – это резолюция цели.

1.3. Как работает интерпретатор Пролога?

Процесс нахождения решения в Прологе заключается в сопоставлении предиката цели с предикатами базы знаний. Этот процесс называется ***унификацией.*** Пусть Пролог-системе предъявлена цель из предыдущего подраздела (мы хотим найти, чьей внучкой является Кристина):

parent(X, kristina), parent(Y, X).

Пролог выделяет из нее первую подцель *parent(X, kristina)* и начинает сопоставлять ее с базой знаний (проводить унификацию). База знаний повторена ниже.

parent(boris, alla).
parent(bedros, filipp).
parent(edmuntas, kristina).
parent(alla, kristina).
parent(kristina, denis).

Вначале сопоставляются первый предикат и подцель:

parent(boris, alla) и *parent(X, kristina)*

Первый аргумент boris сопоставляется с переменной *X*.

Следует иметь в виду, что в Прологе различаются состояния переменных *free* (свободные) и *bound* (связанные). **Если обе переменные связаны, то при унификации происходит их сравнение. Если одна из них свободна, то происходит присвоение. Переприсвоение значений переменным не допускается.** Это существенно отличает Пролог от прочих языков.

Таким образом, переменной *X*, которая пока является свободной, присваивается значение *boris*. После этого унифицируются вторые аргументы, *alla* и *kristina*. Поскольку это константы, и *alla* не равно *kristina*, то унификация предиката *parent(boris,alla)* и подцели *parent(X, kristina)* заканчивается неудачей (*fail*).

Поскольку в базе знаний несколько экземпляров предиката parent, такой предикат называется неоднозначным (*non-derministic*). Если предикат один, то он называется однозначным (*deterministic*).

В случае неоднозначного предиката после неудачи выполняется откат – переход к следующему экземпляру предиката. При этом отменяется также присвоение значение переменным, если таковое имело место. Затем выполняется унификация предикатов

parent(bedros, filipp) и *parent(X, kristina)*

Очевидно, что результат унификации будет тот же, неудача (*fail*). При откате на следующий предикат *parent(edmuntas,kristina)* картина будет иная: *X*

присвоится значение *edmuntas*, а сопоставление вторых аргументов будет также успешным, так как *kristina* = *kristina*. Таким образом, первая подцель окажется выполненной. Пролог запоминает, какой экземпляр предиката сработал и устанавливает на следующий предикат указатель отката:

parent(boris, alla).
parent(bedros, filipp).
parent(edmuntas, kristina).
> *parent(alla, kristina).*
parent(kristina, denis).

после чего перейдет ко второй подцели *parent(Y, X)*, где *X* = *edmuntas*, т.е. Пролог ставит себе такую подцель:

parent(Y, edmuntas).

В поисках родителя Эдмунтаса Пролог снова начинает унифицировать этот предикат с начала базы знаний, начиная с *parent(boris, alla)*. Нетрудно видеть, что на этот раз перебор всех предикатов закончится неудачей, т.е. подцель

parent(Y, edmuntas)

не дала положительного решения. В этом случае Пролог откатывается к предыдущей подцели (продвигается назад по списку подцелей) и пытается найти альтернативное решение для *parent(X, kristina)*. При этом *X* опять становится свободной переменной, а Пролог возвращается к точке отката:

parent(boris, alla).
parent(bedros, filipp).
parent(edmuntas, kristina).
> *parent(alla, kristina).*
parent(kristina, denis).

то есть к предикату *parent(alla, kristina),* сопоставляя его с подцелью *parent(X, kristina)*.

Теперь *X* присваивается значение *alla*, указатель отката устанавливается на предикат *parent(kristina, denis)* и опять выполняется переход к следующей подцели *parent(Y, X)*, где *X* = *alla*. Пролог снова начинает унификацию подцели *parent(Y, alla)* с базой знаний, начиная с первого предиката. В первом предикате происходит унификация константы *boris* и свободной переменной *Y*. Происходит присвоение *Y=boris,* затем сопоставляются вторые аргументы. Так как *alla* = *alla*, сопоставление завершается успешно.

Таким образом, решение найдено: Кристина является внучкой Бориса. Заметим, что неудача, которая постигла нас в поисках деда Кристины по отцовской линии, связана только с неполнотой базы знаний.

Итак, интерпретатор Пролога автоматически выполняет поиск решения. Механизм поиска реализован с помощью отката после неудачи. Откат происходит на следующий экземпляр неоднозначного предиката. Выполнение программы на Прологе (резолюция цели) заключается в унификации цели с базой знаний.

1.4. Факты и правила в Прологе

Описанный выше запрос, устанавливающий отношение типа "прародитель-внук" может потребоваться в дальнейшем неоднократно. В этой связи его целесообразно запомнить для дальнейшего использования в других запросах. В базе знаний Пролога можно хранить не только факты, но и правила, т.е. условные отношения. Отношение типа "прародитель-внук" может быть записано следующим образом:

grandparent(X,Y) if parent(X,Z), parent(Z,Y).

Читать это нужно следующим образом: *X* является прародителем *Y*, если *X* является родителем *Z* и *Z* является родителем *Y*. Предикат *grandparent(X,Y)* называется заголовком правила, а выражение справа от *if* – телом правила.

Примечание: Синонимом связки "if" в правиле являются символы ":-".

Таким образом, как и в базах данных, в базе знаний Пролога в виде фактов мы храним первичные знания, а производные от них записываем в виде правил, к которым обращаемся так же, как и к фактам.

Факт – это то, что известно.

Правило – это способ порождения новых фактов на основе имеющихся.

Для родственных отношений мы можем установить множество правил, избавляясь от необходимости вводить дополнительные факты, например, кто кому приходится братом, племянником и т.д. Правило, определяющее отношение брат (сестра):

sibling(X,Y) :- parent(Z,X), parent(Z,Y), X<>Y.

Предикат сравнения *X<>Y* нужен для разрешения коллизии типа "сын моего отца, но мне не брат". Правило, определяющее отношение типа дядя, выглядит следующим образом:

uncle(X,Y) :- parent(Z,Y), sibling(X,Z).

Когда в ходе резолюции цели Пролог встречает не факт, а правило, то вначале унифицирует заголовок правила, т.е. сравнивает связанные переменные и присваивает значения свободным переменным. В случае успешной унификации аргументов Пролог подставляет значения аргументов из заголовка в первый предикат в теле правила и ставит этот предикат себе в качестве подцели, которую начинает унифицировать с базой знаний. В случае успешной резолюции данной подцели Пролог переходит к следующему условию правила. Если унификация этого предиката условия приводит к неудаче, то Пролог выполняет откат к предыдущему условию правила. Этот откат происходит только в том случае, если этот предыдущий предикат является неоднозначным. Поясним это на примере. Зададимся целью найти, кто является прародителем Кристины:

grandparent(Who, kristina).

Получив такую цель, Пролог начинает унифицировать ее с правилом: *grandparent(X,Y) :- parent(X, Z), parent(Z,Y).* Переменная *Who* в предикате цели является свободной переменной и ее унификация с переменной *X* в заголовке правила будет успешной всегда. Следует заметить, что в Прологе все

переменные являются локальными, т.е. существует только внутри правила. Мы могли бы использовать *X* вместо *Who*, и это были бы разные переменные, которые бы унифицировались точно так же. При необходимости создания глобальных переменных используют динамические факты, которые создаются предикатом *assert* и уничтожаются предикатом *retract* или *retractall.*

Далее унифицируются константа *kristina* с переменной *Y*. Поскольку переменные в заголовке правила всегда сначала являются свободными, выполняется присвоение: *Y = kristina.* Поскольку унификация заголовка правила прошла успешно, Пролог углубляется в тело правила и ставит себе в качестве подцели первый предикат тела правила, подставляя переменные, если они связанные:

parent(X, Z).

Переменные X и Z являются свободными, поэтому успешной будет унификация данной подцели с первым же предикатом parent из базы знаний:

X = boris, Z = alla

После этого Пролог переходит ко второму предикату в правиле, подставляя значение X и Y:

parent(alla, kristina).

Резолюция данной подцели дает истину, а значения переменных, присвоенные в ходе унификации, возвращаются Прологом:

Who = X = boris.

Таким образом, в ходе резолюции основной цели Пролог самостоятельно ставит себе подцели, руководствуясь правилами, находящимися в базе знаний. Рассмотрим другой пример:

grandparent(Who, denis).

Аналогично предыдущему примеру Пролог унифицирует заголовок правила *grandparent(X,Y)* и присваивает значение *Y = denis.* Углубляясь в тело правила, Пролог формирует подцель *parent(X, Z).* Данная подцель возвращает, как и в предыдущем примере,

X = boris, Z = alla

Пролог переходит ко второму предикату правила, подставляя в *parent(Z,Y)* значения переменных:

parent(alla, denis).

Пытаясь унифицировать данную подцель, Пролог сопоставляет переменные (*alla, denis*) с первым экземпляром предиката *parent*, терпит неудачу, откатывается к следующему экземпляру и так далее. Поскольку факта *parent(alla, denis)* в базе знаний нет, резолюция данной подцели оказывается неудачной. Следовательно, унификация первого предиката правила значениями *X = boris, Z = alla* является неверной. Поэтому **Пролог выполняет откат к предыдущему условию правила и пытается найти другое решение для подцели** *parent(X, Z)*. При этом отменяется присвоение переменных (*X = boris*, *Z = alla*). Переменные *X* и *Z* вновь становятся свободными. Заметим, что откат здесь возможен только на неоднозначный предикат. Если в цепочке предикатов

внутри правила встречаются как однозначные, так и неоднозначные предикаты, то откат после неудачи выполняется на ближайший неоднозначный предикат.

При первой унификации данного предиката Пролог установил указатель отката на следующий экземпляр факта *parent*:

parent(boris, alla).
> *parent(bedros, filipp).*
parent(edmuntas, kristina).
parent(alla, kristina).
parent(kristina, denis).

При откате Пролог приступает к унификации данного факта и устанавливает указатель отката на третий экземпляр:

parent(boris, alla).
parent(bedros, filipp).
> *parent(edmuntas, kristina).*
parent(alla, kristina).
parent(kristina, denis).

После унификации второго предиката parent с подцелью *parent(X, Z)* Пролог присвоит значения переменным:

X = bedros, Z = filipp

и снова переходит ко второму предикату *parent(Z,Y)*:

parent(bedros,denis).

Очевидно, резолюция и этой подцели завершается неудачей. Пролог снова откатывается к предыдущему предикату правила и к третьему экземпляру факта *parent*. Успешой окажется только унификация четвертого факта: *parent(alla, kristina),* в результате чего мы получим

Who = X = alla.

Так работает интерпретатор Пролога в случае наличия правил в базе знаний. Таким образом,

Факт – знания, основанные на константах (неизменяемые знания).
Правила – знания, которые выводятся на основании фактов.
Набор фактов и правил не содержит в себе алгоритма.
Правила и факты существуют независимо друг от друга.
Объединение правил для вывода результата происходит в ходе резолюции цели.
Переменные в заголовке правила существуют только внутри данного правила.
При откате внутри правила происходит переход к предыдущему неоднозначному предикату в правиле.

1.5. Рекурсии в языке Prolog

Если нас интересует, является ли *X* предком *Y*, то мы должны последовательно ставить цели:

parent(X,Y).

grandparent(X,Y).
grandgrandparent(X,Y).
grandgrandgrandparent(X,Y).

и так далее, где

grandgrandparent(X,Y) :- parent(X,Z), grandparent(Z,Y).
grandgrandgrandparent(X,Y) :- parent(X,Z), grandgrandparent(Z,Y).

Вместо этого Пролог позволяет записать данное правило следующим образом:

predecessor(X,Y) :- parent(X,Y).
predecessor(X,Y) :- parent(X,Z), predecessor(Z,Y).

Здесь имеет место рекурсивный вызов предиката *predecessor.* Рекурсия в Прологе является мощным средством, позволяющим строить очень компактные и эффективные программы.

Рассмотрим использование рекурсии на примере вычисления факториала.

$n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$

Рекурсивное определение факториала: $0! = 1;\ n! = n \cdot (n-1)!$ Программа на прологе, реализующая вычисление факториала будет выглядеть следующим образом:

f(0,1).
f(N,F) :- N1=N-1, f(N1,F1), F=F1·N.

Обратите внимание, что программа, в сущности, состоит не более, чем из рекурсивного математического описания функция факториала, приведенного выше.

Запустим программу в режиме трассировки и зададим цель *f(3, X).* Пролог начинает сопоставлять предикат цели с базой знаний (CALL означает вызов предиката, RETURN – завершение работы предиката, FAIL – неудачу, REDO – откат):

CALL	*f (3, X)*	цель сопоставляется с *f(0,1)*
FAIL		неудача, т.к. $3 \neq 0$
REDO	*f (3, X)*	Происходит откат на следующий экземпляр *f()*
	N= 3,	Входим в тело правила. Присваиваем *N=3*
	N1= 2	Находим *N1=2*
	f (2, X)	Пролог ставит себе подцель
CALL	*f (2, X)*	которая сопоставляется с *f(0,1)*
FAIL		неудача, т.к. $2 \neq 0$
REDO	*f (2, X),*	Происходит откат на следующий экземпляр *f()*
	N= 2,	Опять входим в тело правила. Присваиваем *N=2*
	N1=1	Находим *N1=1.* Это уже другое *N1*
	f (1, X)	Пролог ставит себе подцель *f (1, X)*
CALL	*f (1, X)*	которая сопоставляется с *f(0,1)*
FAIL		*неудача,* т.к. $1 \neq 0$
REDO	*f (1, X)*	Происходит откат на следующий экземпляр *f()*
	N=1	Опять входим в тело правила. Присваиваем *N=1*

	N1= 0	Находим *N1=0.*
	f(0, X)	Пролог ставит себе подцель *f(0, X)*
CALL	*f(0, X)*	которая сопоставляется с *f(0,1)*
RETURN	*X=1*	Успешно. Возврат из нижнего уровня рекурсии
	F=1	Умножение *1* на *1* (факториал от *1*)
RETURN	*X=1*	Возврат из следующего уровня рекурсии
	F=2	Умножение *2* на *1* (факториал от *2*)
RETURN	*X=2*	Возврат из следующего уровня рекурсии
	F=6	Умножение *3* на *2* (факториал от *3*)
RETURN	*X=6*	Возврат из программы

Можно заметить, что логика работы программы вычисления факториала зависит от расположения в тексте предиката, определяющего выход из рекурсии. Унификация выполняется в порядке следования предикатов в тексте программы, и если предикат *f(0,1)* поставить в конце, выход из рекурсии будет невозможен. Таким образом, декларативность Пролога не является абсолютной для удобства его использования. Вариант программы, в котором предикаты могут располагаться в любом порядке, представлен ниже.

```
f(N,F) :- N>0, N1=N-1, f(N1,F1), F=F1·N.
f(0,1).
```

Заметим также, что в данном рекурсивном предикате есть действия в теле правила после рекурсивного вызова. Это называют нарушением хвостовой рекурсии (tail recursion).

Нарушение хвостовой рекурсии, называемое также нехвостовой рекурсией, требует запоминания всего окружения рекурсивного вызова (а не только результата), поэтому приводит к большим затратам памяти. Существуют приемы устранения нехвостовых рекурсий. Ниже приведен пример вычисления факториала без нехвостовой рекурсии.

```
f(N1,N,F1,F) :-          % если N1! = F1, то N! = F
 N2=N1+1,
 F2=F1*N2,               %  (N1+1)! = F2
 f(N2,N,F2,F).           % если N2! = F2, то N! = F
 f(N,N,F,F).             % Условие выхода из рекурсии
```

Цель для вычисления 3! выглядит следующим образом: *f(0,3,1,F).*

Рекурсия в Прологе не всегда используется для выполнения многократно повторяющихся действий. Вспомним базу знаний «звездной» семьи, в которой есть предикат, описывающий супружеские отношения, в частности, *spouse(filipp,alla).*

Супружеские отношения в отличие от родительских отношений, являются симметричными. Филипп является супругом Аллы, равно как и Алла является супругой Филиппа. При этом в предикате положение аргументов является фиксированным. Иными словами, если факт в базе знаний записан следующим образом:

spouse(filipp,alla).

а предикат цели таким:

spouse(alla,filipp).

то результат будет отрицательным. Для того, чтобы показать Прологу, что это предикат является симметричным в отношении аргументов, мы можем применить правило:

spouse(X,Y) :- spouse(Y,X).

В качестве примера рассмотрим известную коллизию. В деревне жили две семьи: мать с дочерью и отец с сыном. Дадим им имена. Пусть мать и дочь зовут Мария и Даша, а отца и сына Олег и Сергей соответственно. Первые буквы имен подскажут нам, кто есть кто, иначе мы запутаемся. В базе знаний эти факты найдут свое отражение в следующем виде:

parent(oleg, sergei).
parent(maria, dasha).

В силу превратностей судьбы Олег женился на Даше, и Мария вышла замуж за Сергея:

spouse(oleg,dasha).
spouse(sergei, maria).

Для симметрии супружеских отношений введем правило:

spouse(X, Y) :- spouse(Y, X).

Нравы в деревне простые, поэтому жену отца положено называть мамой, а мужа матери – отцом. Правила для этого будут такими:

parent(X,Y) :- spouse(X,Z), parent(Z,Y).

Попробуем найти внука Сергею. Ставим цель

grandparent(sergei, Who).

Пролог находит правило *grandparent(X,Y) :- parent(X,Z), parent(Z,Y)* и ставит себе подцель из первого предиката в теле этого правила:

parent(sergei, Z).

У Сергея детей нет, поэтому Пролог обращается к правилу *parent(X,Y) :- spouse(X, Z), parent(Z,Y)* и ставит себе цель

spouse(sergei, Z).

Пролог дает результат *Z = maria*. Второй предикат правила *parent*

parent(maria,Y).

Возвращается значение *Y = dasha*. То есть Сергей в качестве мужа Марии числится отцом Даши. Теперь Пролог переходит ко второму предикату в правиле *grandfather*:

parent(dasha,Y).

Даша также не имеет детей, поэтому Пролог обращается к правилу *parent(X,Y) :- spouse(X,Z), parent(Z,Y)* и сначала пытается найти супруга Даше:

spouse(dasha, Z).

Результат: *Z = oleg*. Второй предикат этого правила *parent(oleg,Y)* даст *Y = sergei.* То есть Сергей приходится внуком самому себе! Созданная нами база знаний верно отражает данную коллизию.

1.6. Рекурсии и итерации

Напишем на Прологе простую программу, которая имитирует на компьютере пишущую машинку:

type :- readchar(X), write (X), type.

Стандартный предикат *readchar* выполняет чтение символа с клавиатуры. Это бесконечная рекурсия – из нее нет выхода. Модифицируем программу таким образом, чтобы она обеспечивала ввод только одного предложения (до первой точки):

type :- readchar(X), write(X), X <> '.', type.
type.

Первый предикат *type* читает символ, выводит его на экран, после чего сравнивает его с точкой. Если введенный символ не точка, то происходит рекурсивный вызов *type*, в противном случае – откат на следующий, пустой предикат *type*. Второй предикат *type* нужен лишь для того, чтобы вся программа завершилась успехом, а не неудачей.

Заметим, что рекурсия в этой программе совершенно не нужна, так как вызов предиката из самого себя не имеет никакого смысла для логики программы. По программистской привычке нам было бы предпочтительней зациклить данное правило. Но в Прологе GOTO нет в принципе. Вместо этого есть возможность вызывать откат. Напишем следующую конструкцию:

type :- readchar(X), write(X), X = '.'.

При первом выполнении данного правила (введенный символ не равен точке) предикат завершится неудачей. Отката же не будет по той причине, что все предикаты в теле правила являются однозначными. Кстати, ***все стандартные предикаты являются однозначными.***

Таким образом, для того, чтобы откатиться на начало правила, необходимо, чтобы там был неоднозначный предикат. Создадим такую конструкцию:

repeat.
repeat :- repeat

Предикат *repeat* ничего не делает кроме того, что является неоднозначным. Можно даже сказать, очень неоднозначным, поскольку перебрать все его варианты невозможно. Это бесконечная рекурсия.

Программа, имитирующая пишущую машинку, без рекурсивного вызова type будет выглядеть следующим образом:

repeat.
repeat :- repeat
type :- repeat, readchar(X), write(X), X = '.'.

Здесь рекурсия заменена итерацией, хотя рекурсивный вызов в виде предиката repeat все же присутствует.

1.7. Отсечения в Прологе

Из всего вышесказанного можно сделать справедливый вывод о том, что интерпретатор Пролога, выполняя резолюцию цели, всегда делает полный обход дерева решений. Спуск по дереву вниз соответствует углублению в тело правила, возврат наверх и переход на соседнюю ветвь – откату после неудачи. Рассмотрим концепцию отсечения на конкретном примере.

Пусть нас пригласили на дачу. При этом обычно дают подробную инструкцию, как проехать и найти конкретный дом. Например, полученная нами инструкция выглядит следующим образом:

1. Въехать в деревню Васино (а может быть и Ванино. Написано неразборчиво).
2. Повернуть направо.
3. Доехать до колодца.
4. Искомый дом из красного кирпича – первый от колодца по правой стороне.

Запишем правило поиска дачи предикатами Пролога:

```
dacha1(X) :-
        enter_village(X),
        find_a_house.
find_a_house :-     turn_right,
        meet_mine,
        see_a_red_brick_house.
enter_village(vasino).
enter_village(vanino).
```

Начинаем поиск (рис.1.1)...

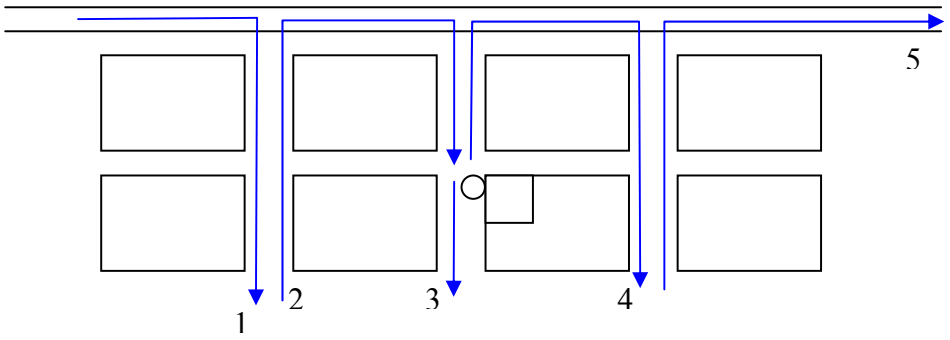


Рис.1.1. Иллюстрация отсечения в Прологе

Въезжаем в деревню и поворачиваем направо (стрелка 1). Проезжаем до конца деревни и не находим колодца. Выполняем откат, то есть возвращаемся на шоссе и едем до следующего поворота направо (стрелка 2), доезжаем до колодца и видим, что ближайший к нему дом из белого кирпича. Едем до конца улицы и видим, что колодцев больше нет (стрелка 3). Откатываемся на шоссе и едем до следующего поворота (стрелка 4), проезжаем всю улицу и

откатываемся, поскольку колодцев здесь нет (стрелка 5). Из этого делаем вывод, что название деревни мы прочитали неверно, и дачу следует искать в следующей деревне. В деревне Ванино повторяем все сначала.

Отсечение позволяет сократить количество пробегаемых ветвей дерева решений. Если хозяин дачи даст нам важную дополнительную информацию о том, что в каждой деревне есть только один колодец (а если в качестве ориентира заданы почта или милиция, то можем догадаться сами, что они могут быть в деревне только в одном экземпляре), мы не будем выполнять действия, обозначенные стрелками 3 и 4, а сразу поймем, что попали не в ту деревню.

Отсечение – это предикат, который обозначается восклицательным знаком и вставляется в правило. Срабатывает отсечение после того, как Пролог пройдет через него. Отсечение уничтожает указатели отката, установленные в ходе унификации данного правила, т. е. делает предыдущие предикаты данного правила однозначными.

Если мы хотим указать в программе поиска дачи, что колодец в деревне единственный, мы должны поставить отсечение после найденного колодца:

```
dacha2(X) :-
        enter_village(X),
find_house.
find_a_house :-    turn_right,
        meet_mine, !,
        see_a_red_brick_house.
enter_village(vasino).
enter_village(vanino).
```

Как только Пролог достигнет отсечения, весь предикат *find_house* станет однозначным, следовательно, дальнейшая неудача в предикате *see_a_red_brick_house* приведет к тому, что предикат *find_house* также завершится неудачей. Если при этом предикат *enter_village* является неоднозначным (как показано в программе), то мы имеем шанс найти дачу в другой деревне, поскольку отсечение в правиле *find_house* не затрагивает свойства предиката *dacha* и установленные в нем указатели отката.

Если предикат *enter_village* является однозначным, то нам придется возвращаться домой. Дачи мы не нашли. Если дом возле колодца окажется все-таки из красного кирпича, то наличие отсечения никак не скажется на результате поиска. Таким образом, отсечение следует включать в программу на Прологе всегда, когда нам точно известно, что решение либо единственное, либо его нет вовсе.

Примечание. Если программа поиска дачи будет выглядеть следующим образом:

```
dacha3(X) :-
        enter_village(X),
        turn_right,
        meet_mine, !,
```

see_a_red_brick_house.
enter_village(vasino).
enter_village(vanino).

то отсечение, вставленное в указанном месте, приведет к тому, что все предыдущие предикаты в правиле *dacha* станут однозначными, в том числе и *enter_village.* Иными словами, нам будет запрещено искать дачу в следующей деревне.

1.8. Красное и зеленое отсечения

Если отсечение не влияет на логику работы программы, а только сокращает возможные спуски по ложным ветвям дерева решений, то такое отсечение называется зеленым.

Если отсечение влияет на логику программы (без него программа работает иначе), такое отсечение называется красным.

В приведенном выше правиле *dacha2* отсечение зеленое, поскольку не влияет на логику программы, а только избавляет нас от бесполезного блуждания по деревне. В правиле *dacha3* отсечение меняет логику программы, т.к. поиск ограничивается первым найденным колодцем на всем пути следования к цели, а не в каждой деревне. Следует заметить, что и зеленое отсечение в правиле *dacha2* все же меняет логику программы. Правда, это отсечение отсекает лишь возможность поехать на две дачи сразу.

Использование отсечений позволяет существенно сокращать как время работы программы на Прологе, так и объем требуемой оперативной памяти. Кроме того, вдумчивая расстановка отсечений сокращает отладку программ, поскольку устраняет коллизии, подобные той, что приведена в следующем примере.

Пусть человек считается уважаемым в обществе, если он/она состоит в браке и имеет свой дом. Правило для этого выглядит следующим образом:

respectable(X) :- spouse (X,_),
has_home(X).

Пусть в базе знаний имеются следующие факты и правила:

spouse(alla, filipp).
spouse (X,Y) :- spouse (Y, X).
has_home(alla).

Зададим следующую цель: *respectable(filipp).*

Для резолюции данной цели Пролог войдет в тело правила *respectable* и поставит себе подцель: *spouse (filipp,_).*

Поскольку такого факта в базе нет, Пролог выполнит откат на правило

spouse (X,Y) :- spouse (Y, X).

из которого создаст себе подцель: *spouse (_, filipp).*

В ходе резолюции данной подцели будет получено успешное решение, поскольку такой факт в базе есть. Далее Пролог ставит себе следующую

подцель из правила *respectable*: *has_home(filipp).*

Такого факта в базе нет, и, поскольку, предыдущий предикат правила *respectable***,** то есть *spouse* является неоднозначным, то выполняется откат на следующий экземпляр этого предиката, а именно:

spouse (X,Y) :- spouse (Y, X).

Смысл отката заключается в поиске другой жены для Филиппа. Будет поставлена следующая подцель:

spouse (filipp, _).

Таким образом, будет происходить бесконечное углубление в рекурсию. Совершенно очевидно, что, найдя одну жену Филиппа, мы должны сообщить Прологу, что это решение является единственным, и искать другие не нужно.

```
respectable(X) :-   spouse (X,_), !,
                    has_home(X).
spouse(alla, filipp) :- !.
spouse (X,Y) :- spouse (Y, X).
has_home(alla).
```

1.9. Списки в Прологе

Все переменные, с которыми мы работали ранее, были скалярами. Теперь рассмотрим агрегаты данных в Прологе. Один из них – список. Список – это упорядоченная последовательность однотипных элементов. Элементы списка заключаются в квадратные скобки и разделяются запятыми:

```
[ 1, 2, 3, 4, 6, 7, 8]                  – integer
[mon,tue, wed, thu, fri, sat, sun]      – symbol
["Иванов", "Петров"]                    – string
[1.5, 2.22, 0.001, 0]                   – real
[]                                      – пустой список
```

Объявляются списки следующим образом:

```
domains
    sym = symbol*          % список символьных значений
    intlist = integer*     % список целых
    realist = real*        % список действительных чисел
```

Единственная операция, которая допускается над списком – это отсечение головы от хвоста, причем «разместить» эту операцию можно только в аргументах предикатов.

Напишем некоторые полезные предикаты для работы со списками. В частности, как определить является ли некоторая переменная элементом списка? Разобьем список на голову и хвост. Искомое значение находится либо в голове списка, либо в хвосте:

```
member (H, [H | _] ).
member (X, [H | T] ) :- member (X, T).
```

Вышеописанным приемом можно искать не только один элемент, но и больше (например пару последовательных значений в списке)

memb2(H1, H2, [H1, H2 | _]).
memb2(H1, H2, [H | T]) :- memb2(H1, H2, T)

Чтобы включить значение в список (проще всего в голову) можно составить такое правило: *incl (H, T, [H | T]).*

Исключить значение из списка несколько сложнее:

excl (H, [H | T], T). *% исключаем из головы*
excl (X, [H,T], [H | TT]) :- excl (X, T, TT). % исключаем из хвоста

Ниже приведены другие примеры предикатов работы со списками:

1. Программа вывода на экран элементов списка по одному:

print_list([]). % выход из рекурсии
print_list([H | T]) :- write(H, " "), % вывод головы списка и пробела
print_list(T). % вывод хвоста

Если задать цель

printlist(["я", "помню","чудное","мгновенье"]).

то результат будет следующим:

я помню чудное мгновенье

2. Та же программа, но выводящая список в обратном порядке:

print_inverse([]). % выход из рекурсии
print_inverse ([H | T]) :- print_inverse (T), % вывод хвоста
write(H, " "). % вывод головы списка и пробела

Цель:

write_inverse (["я", "помню","чудное","мгновенье"]).

Результат:

мгновенье чудное помню я

3. Программа, подсчитывающая сумму элементов списка:

sum([], 0). % выход из рекурсии,
sum([H | T], S) :- sum(T,S1), % S1 – сумма элементов в хвосте списка
S = S1 + H. % S – остается добавить к сумме только голову

Цель:

sum([1,2,3,4,5,6], S), write ("Сумма =",S).

Результат: *Сумма=21*

4. Программа, подсчитывающая количество элементов списка:

count([], 0). % выход из рекурсии,
count([H | T], S) :- count(T,S1), % S1 – счетчик элементов
% в хвосте списка
S = S1 + 1. % S – остается добавить к сумме единицу

Цель:

count([1,2,3,4,5,6], S), write ("Количество =",S).

Результат: *Количество =6*

1.10. Пример: Решение логической задачи о волке, козе и капусте

Рассмотрим известную логическую задачу о волке, козе и капусте. Фермер

должен переправить на другой берег реки волка, козу и капусту. Грузоподъемность лодки такова, что за один раз можно взять на борт что-нибудь одно: или волка, или козу, или капусту. В присутствии старика никто никого не ест. Если же он отлучится, то волк съест козу, а коза – капусту.

Для решения этой задачи организуем два списка. Один список будет отражать содержимое левого берега, второй – правого. Первоначально все находятся на левом берегу. Список левого берега: [wolf, goat, cabbage], список правого берега пустой: [ ].

Определим предикаты, описывающие задачу.

```
stuff(wolf).          % перечисление груза
stuff(goat).
stuff(cabbage).
/*  условия возникновения конфликта       */
conflict(X): - member(wolf, X), member(goat, X).
conflict(X); - member(goat, X), member(cabbage, X).
 /*      Предикаты, описывающие перемещения лодки:
 С левого берега на правый                */
go_right([ ], _).    %  условие завершения (список левого берега пуст)
go_right( L, R) :-
        stuff(X),                  % выбираем груз,
member(X, L),                      % который есть на левом берегу
excl(X, L, LL),                    % исключаем из списка левого берега
 not(conflict(LL)),                % на левом берегу конфликта нет
 incl(X,R,RR),                     % включаем в список правого берега
write(LL,"--",X,"-->",R),          % выводим сообщение
go_left(LL,RR).                    % гребем назад
```

/* Движение влево возможно в двух вариантах. Если на правом берегу конфликт не возникает, то фермер едет один */

```
go_left(L,R) :- not(conflict(R)),
write(L,"<-------",R),       % выводим сообщение
go_right(L, R).              % вызываем предикат движение вправо
```

/* Если на правом берегу возникает конфликт надо кого-нибудь увезти обратно. Это единственная подсказка, которую мы сообщаем программе. В остальном Пролог будет искать решение вполне самостоятельно */

```
go_left(L,R) :-
        stuff(X),                  % выбираем груз,
        member(X, R),              % который есть на правом берегу
        excl(X, R, RR),            % исключаем из списка правого берега
        not(conflict(RR)),         % на правом берегу конфликта нет
        incl(X,L,LL),              % включаем в список левого берега
        write(L,"<--",X,"--",RR),        % выводим сообщение
        go_right(LL,RR).           % гребем назад
```

Вызов цели должен быть следующим:

```
        go_right([wolf, goat, cabbage], [ ]).
```

Если запустить эту программу, то быстро обнаружится, что первым рейсом старик отвезет на правый берег козу. Вторым рейсом – волка. Оставить волка с козой на правом берегу нельзя, поэтому он отвезет первый попавшийся груз, а им окажется волк, обратно. И так будет возить его бесконечно.

Недостаток данной программы заключается в том, что фермер не помнит, кого он только что привез. Для укрепления его памяти добавим в предикаты *go_left* и *go_right* название последнего перевезенного груза. Финальный вариант программы выглядит следующим образом (изменения выделены полужирным шрифтом):

```
        /*      Задача о волке, козе и капусте                  */
domains   % раздел требуется для PDC-Пролога. В SWI-Прологе не нужен
        stuff = wolf; goat; cabbage; nil  % создаем свой тип данных ( nil – пустое )
        list  = stuff*                    % тип данных список
predicates   % раздел требуется для PDC-Пролога. В SWI-Прологе не нужен
          member(stuff,list)
          incl(stuff,list,list)
          excl(stuff,list,list)
          conflict(list)
          go_right(list, list, stuff)
          go_left(list, list, stuff)
clauses
        stuff(wolf).
        stuff(goat).
        stuff(cabbage).
        member (H, [H | _] ).
        member (X, [H | T] ) :- member (X, T).
        incl (H, T, [H | T]).
        excl (H, [H | T], T ).
        excl (X, [H,T], [H | TT] ) :- excl (X, T, TT).
        conflict(X): - member(wolf, X), member(goat, X).
        conflict(X); - member(goat, X), member(cabbage, X).
        go_right( L, R,Last) :-
                stuff(X),                 % выбираем груз,
                X <> Last,                % но не тот, что везли только что и
                member(X, L),             % который есть на левом берегу
                excl(X, L, LL),           % исключаем из списка левого берега
                not(conflict(LL)),        % на левом берегу конфликта нет
                incl(X,R,RR),             % включаем в список правого берега
                write(LL,"--",X,"-->",R),       % выводим сообщение
                go_left(LL,RR,X).         % гребем назад
        /*      Если на правом берегу конфликт не возникает, то едет один */
        go_left(L, R, Last) :- not(conflict(R)),
        write(L,"<-------",R),       % выводим сообщение
```

```
go_right(L, R, nil).        % вызываем предикат движение вправо
                            % nil – значит, не везли ничего
```

/* Если на правом берегу возникает конфликт надо кого-нибудь увезти обратно, но не того, кого везли только что*/

```
go_left(L,R,Last) :-
        stuff(X),           % выбираем груз,
        X <> Last,          % не тот, что везли только что и
member(X, R),               % который есть на правом берегу
excl(X, R, RR),             % исключаем из списка правого берега
not(conflict(RR)),          % на правом берегу конфликта нет
incl(X, L, LL),             % включаем в список левого берега
write(L,"<--",X,"--",RR),   % выводим сообщение
go_right(LL, RR, X).        % гребем назад и помним, что везли X

goal
        go_right(([wolf, goat, cabbage], [ ], nil).
/* Конец программы    */
```

1.11. Контрольные вопросы

1. Напишите правило для отношения двоюродный брат или сестра. Используйте правила, приведенные в подразд.1.4.
2. Имеется программа, состоящая из двух экземпляров правила *a*:

a :- b, c, !, d, fail.
a :- e.

Какие предикаты будут вызваны при выполнении правила *a*, если каждый из предикатов *b, c, d, e* заканчивается успехом, а предикат *fail* — стандартный предикат для искусственного вызова неудачи? Что изменится, если убрать предикат отсечения?

3. Имеется предикат

dosomething([]).
dosomething([H|T]) :- member(H,T), dosomething(T).
dosomething([H|T]) :- write(H), dosomething(T).

Какой результат даст вызов *dosomething([1,0, 0,1, 2,0,10])*?

4. Какое отсечение, красное или зеленое, стоит в следующем правиле. Обоснуйте.

sister(X,Y) :- sibling(X,Y), !, female(X).

5. Правомерно ли использование отсечения в следующем правиле? Почему?

grandfather(X,Y) :- father(X,Z), !, parent(Z,Y).

2. Решение проблем методом поиска

2.1. Что такое метод поиска

Рассмотренная в предыдущем разделе задача о волке, козе и капусте является типичным примером задачи поиска. В примере решения задачи осознанно использовался «программистский подход». На самом деле такие задачи достаточно просто формализуются к единому шаблону. Они характеризуются наличием дискретных состояний, одно из которых является начальным, другое – конечным, а также заданной логикой перехода из одного состояния в другое. Следовательно, для их решения достаточно описать начальное и конечное состояния, а также алгоритм перехода из одного состояния в другие (выбор преемника). В зависимости от того, важен ли путь к цели, может потребоваться запоминание пройденных состояний. Например, для задачи о фермере важен только путь к цели, поскольку конечное состояние задано. В других задачах наоборот, не важен путь, а нужно конечное состояние. Пример – задача о восьми ферзях, где требуется на шахматной доске расставить так, чтобы ни один из них не бил другого.

Преобразуем задачу про волка, козу и капусту в новый вид. Обозначим четырьмя булевыми переменными наличие или отсутствие на левом берегу волка, козы, капусты и лодки соответственно (начальное состояние 1,1,1,1). Необходимо переместить всех на правый берег (конечное состояние 0,0,0,0). Описывать то, что находится на правом берегу нет необходимости. То, чего нет на одном берегу, есть на другом. С учетом ограничений (волка нельзя оставлять с козой, а козу – с капустой) все допустимые переходы можно отобразить в виде графа состояний (рис.2.1.). Для записи пути к цели введем еще пару переменных типа «список». В первой будет накапливаться последовательность пройденных вершин дерева решений, последняя будет использоваться для возврата результата.

Запишем это на Прологе:

```
farmer(0,0,0,0,Path,Path).        % конечное состояние – все на правом берегу
farmer(W,G,C, F, P, Path) :-      % W,G,C,B - текущее состояние
      next(W,G,C,F, W1,G1,C1,F1),          % выбор преемника
      farmer(W1,G1,C1,F1 [[W,G,C,B] | P],Path).   % переход к следующему
                                                  % состоянию
next(W,G,C,1, W,G,C,0)  :-                % выбор преемника
     move(W,G,C,F, W1,G1,C1,F1), not(conflict(W,G,C,B)).
move (1,G,C,1,  0,G,C,0).         % везем волка вправо
move (W,1,C,1, W,0,C,0).          % везем козу вправо
move (W,G,1,1, W,G,0,0).          % везем капусту вправо
move(W,G,C,1, W,G,C,0).           % идем порожняком вправо
move (W,G,C,0, W,G,C,1).          % идем порожняком влево
move (0,G,C,0,  1,G,C,1).         % везем волка влево
```

move (W,0,C,0, W,1,C,1). % везем козу влево
move (W,G,0,0, W,G,1,1). % везем капусту влево

% конфликтные состояния

conflict(1,1,_,0). % волк и коза на левом берегу, фермер – на правом
conflict(_,1,1,0). % коза и капуста на левом берегу, фермер – на правом
conflict(0,0,_,1). % волк и коза на правом берегу, фермер – на левом
conflict(_,0,0,1). % коза и капуста на правом берегу, фермер – на левом

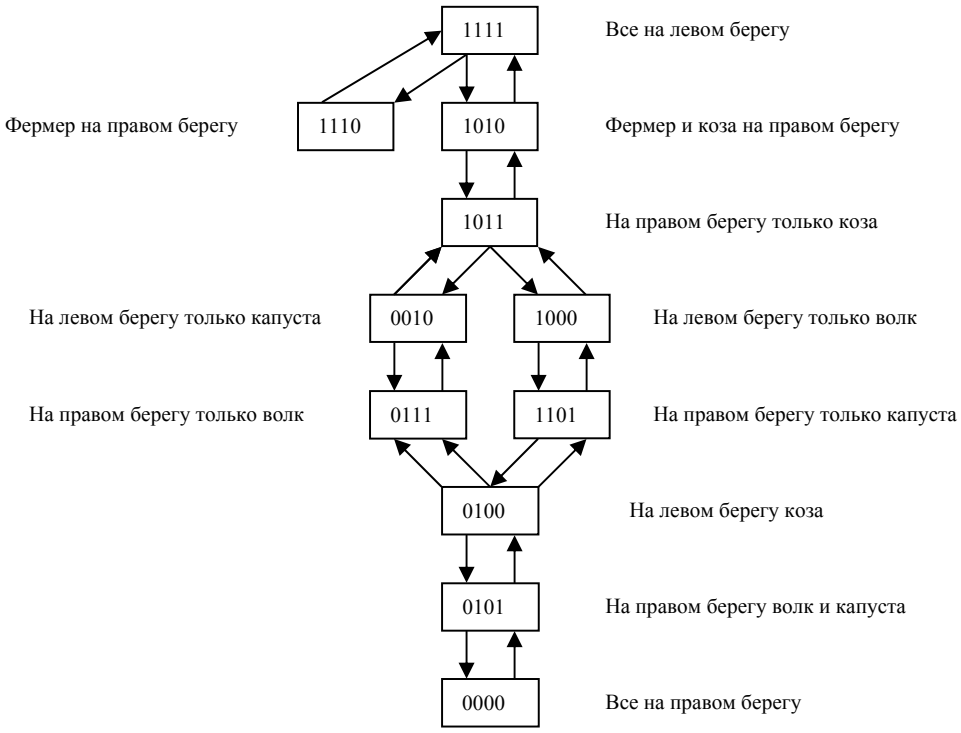


Рис.2.1. Граф состояний для задачи о волке, козе и капусте

Изобразим дерево решений для данной программы (рис.2.2). Поскольку Пролог начинает унифицировать предикаты в порядке их следования в программе, он реализует поиск методом «сначала вглубь» (поиск в глубину).

Даже не запуская программу, мы видим, что, спускаясь по левой ветви дерева решений, Пролог впадает в бесконечную рекурсию. При этом фермер катается порожняком с одного берега на другой. В этом состоит главный недостаток метода поиска в глубину: если есть бесконечные ветви дерева, то поиск не может быть завершен.

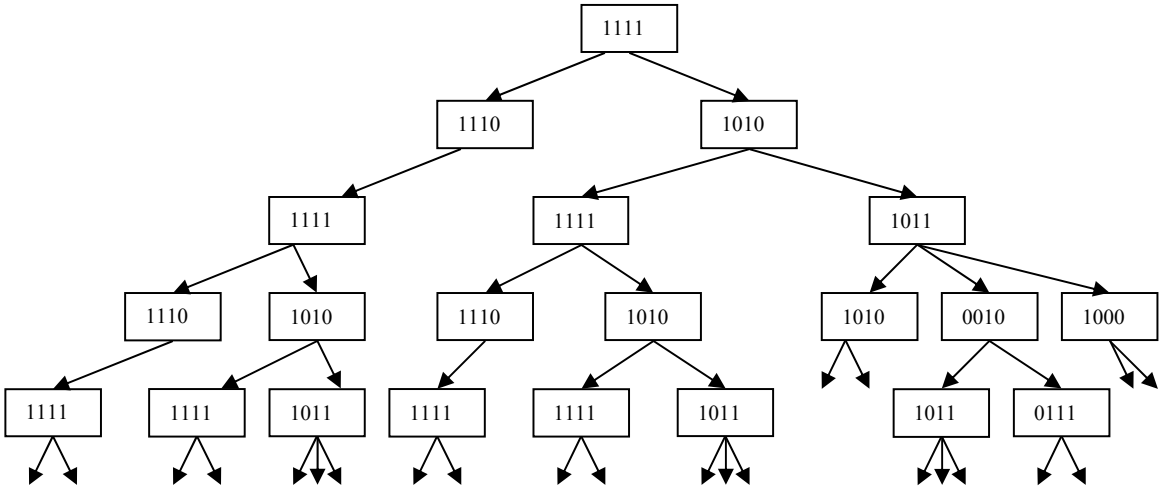


Рис.2.2. Дерево решений задачи о волке, козе и капусте

Модификация данного метода путем исключения повторного перехода на уже посещенные узлы позволяет избежать бесконечного спуска по дереву:

```
farmer(W,G,C, F, P, Path) :-          % W,G,C,B - текущее состояние
    next(W,G,C,F, W1,G1,C1,F1),       % выбор преемника
    not(member([W,G,C,B], P)),        % исключение повторного посещения
    farmer(W1,G1,C1,F1 [[W,G,C,B] | P],Path).   % переход к следующему
                                                % состоянию
```

Если теперь задать предикат цели

```
farmer(1,1,1,1,[],Path).
```

то переменная *Path* будет содержать список состояний от начального до конечного по кратчайшему пути к цели.

2.2. Неинформированный поиск

Рассмотренный выше пример демонстрирует неинформированный (слепой) поиск, при котором отсутствует информация о том, в правильном ли направлении он ведется. Такой поиск представляет собой комбинаторную задачу, размерность которой определяется коэффициентом ветвления (b) и глубиной (d) по формуле b^{d+1}. В худшем случае при поиске нужно развернуть $b^{d+1} - 1$ узлов (сам целевой узел не развертывается), а в среднем – $b^{d+1} / 2$ узлов.

Размерность даже не очень сложных комбинаторных задач поражает воображение. Например, для оценки алгоритмов поиска часто используют упомянутую выше задачу о восьми ферзях, которые надо так расставить на шахматной доске, чтобы ни один из них не бил другого. Пространство состояний для такой задачи составляет приблизительно $3*10^{14}$ узлов. При развертывании 10000 узлов в секунду на обход всего дерева потребуется приблизительно 1000 лет! В то же время человек в состоянии решить данную

задачу за считанные минуты. Рассмотрим, какие существуют варианты и методы улучшения неинформированного поиска.

Поиск в ширину. Вначале развертывается корневой узел, затем все ее преемники, затем – преемники преемников и т.д. В отличие от поиска в глубину этот метод быстро обнаруживает решение, которое находится неглубоко, и не впадает в бесконечные углубления. Существенным недостатком данного метода является необходимость запоминать все узлы-предки, количество которых также определяется формулами комбинаторики. Для упомянутой задачи о восьми ферзях при расходе памяти 1000 байт на вершину объем требуемой памяти измеряется сотнями петабайтов (1 терабайт = 1024 гигабайта, 1 петабайт = 1024 терабайта). Очевидно, что этот недостаток существенно ограничивает применимость метода.

Поиск по критерию стоимости. В этом случае вначале развертывается узел с наименьшей стоимостью пути. Данный поиск применяется, если стоимости переходов от узла к узлу различны. При одинаковых стоимостях метод вырождается в поиск в ширину. Поиск по критерию стоимости дает хорошие результаты, хотя и не свободен от бесконечных блужданий.

Поиск с ограничением глубины. Идея метода заключается в том, что заранее оценивается предельная глубина дерева, и спуск по ветви дерева прекращается по достижении данной глубины. Это помогает предотвратить бесконечный спуск и длительные бесцельные блуждания. Недостаток метода – необходимость располагать оценкой сложности задачи. Если установленная предельная глубина будет слишком большой, это повлечет лишние затраты на поиск, если слишком малой – цель не будет достигнута.

Поиск в глубину с итеративным углублением – это поиск с ограничением глубины, которая постепенно увеличивается. Вначале устанавливается глубина 1, затем 2 и т.д. Несмотря на то, что поиск каждый раз начинается сначала, это не приводит к большим затратам. Затраты времени ненамного выше, чем при поиске в ширину, зато не требуется хранить данные о вершинах-предках.

Двунаправленный поиск. Эта разновидность поиска используется в тех случаях, когда конечное состояние нам известно, а нужно найти путь к цели. Один поиск запускается с начального узла, другой – с конечного. Задача завершается, когда оба поиска находят общий узел. Преимущество метода очевидно. Пусть глубина поиска d = 6, а коэффициент ветвления b = 2. Поиск в одном направлении потребует $2^6 = 64$ шага, а двунаправленный поиск $2^{6/2} = 8$ шагов. Недостатком данного метода является необходимость вычислимости узла – предшественника, что бывает не всегда возможным.

Предотвращение формирования повторяющихся состояний. В примере задачи о фермере мы уже использовали эту модификацию метода поиска, правда, там запоминались только узлы текущего пути. В случае ложного пути в этой программе выполняется откат, и Пролог «забывает» уже пройденные узлы. Правильная модификация алгоритма поиска должна включать в список все

пройденные вершины. Следует, однако, заметить, что в некоторых задачах повторяющиеся состояния являются неизбежными. Одной из разновидностей метода предотвращения повторяющихся состояний является использование свойств симметричности. Например, сложность задачи о восьми ферзях или игра в крестики-нолики сокращается в четыре раза, поскольку игровое поле квадратное.

Сравнительные характеристики методов неинформированного поиска сведены в таблицу 2.1.. Здесь используются следующие обозначения:

b – коэффициент ветвления; d – глубина самого поверхностного решения; m - максимальная глубина дерева; e - предел глубины; C – стоимость решения; n – средняя стоимость одного шага.

Таблица 2.1. Характеристики методов неинформированного поиска

Метод	Полнота	Временная сложность	Затраты памяти	Оптимальность
Поиск в ширину	Да	b^{d+1}	b^{d+1}	Да
Поиск по критерию стоимости	Да	$b^{1+C/n}$	$b^{1+C/n}$	Да
Поиск в глубину	Нет	b^{m}	bm	Нет
Поиск с ограничением глубины	Нет	b^{e}	be	Нет
Поиск с итеративным углублением	Да	b^{d}	bd	Да
Двунаправленный поиск	Да	$b^{d/2}$	$b^{d/2}$	Да

2.3. Информированный поиск

В предыдущем разделе было показано, что в условиях отсутствия информации задача поиска может быть решена успешно, но при этом затраты времени и памяти могут быть совершенно неприемлемыми. Если задачу о ферзях попытаться решить на доске 100*100, то количество состояний будет около 10^{400}. Пусть вас не введет в заблуждение компактная экспоненциальная форма представления этого числа. Число атомов в видимой части вселенной составляет около 10^{80}.

В этой связи представляется целесообразным улучшить слепой поиск в тех случаях, когда имеется какая-либо дополнительная информация. Разумеется, мы не рассматриваем случай, когда точно известен путь к цели. В последнем случае задача уже решена. Речь идет об информации, которая наблюдается, но не дает прямого ответа, как двигаться к цели.

Рассмотрим следующую задачу. Пусть нам нужно добраться из Таллина в

Москву воздушным транспортом. Расстояния между городами и воздушные трассы показаны на рис.2.3.

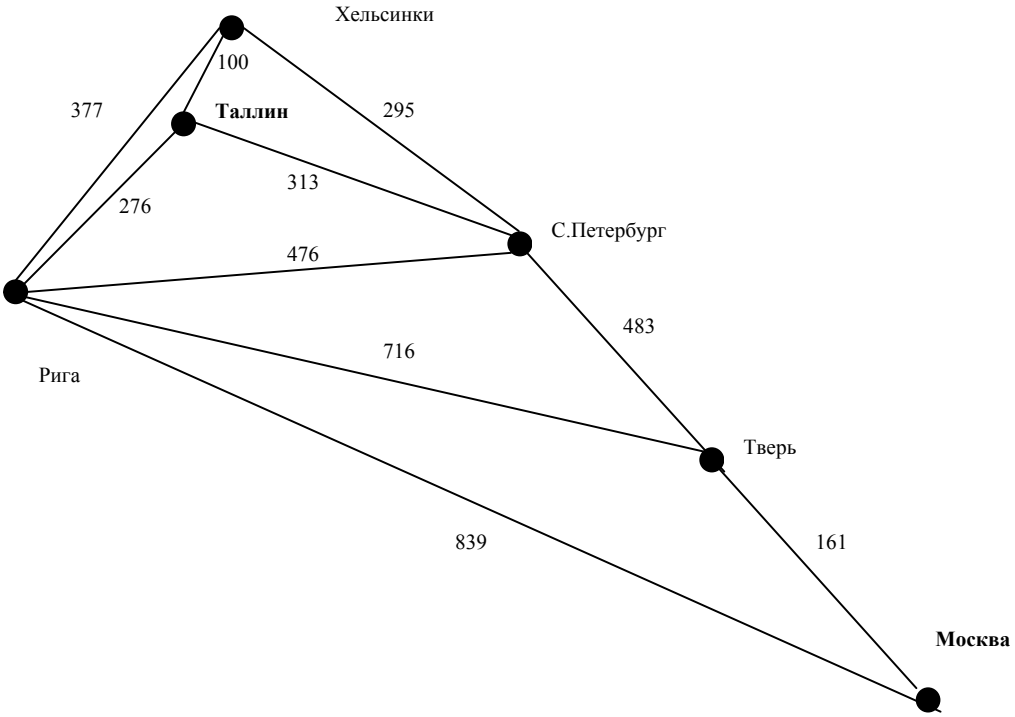


Рис.2.3. Пример воздушных трасс к задаче поиска

Если бы мы не располагали никакими данными о расстояниях, то имел бы место слепой (неинформированный) поиск. Если мы знаем только расстояние от текущей точки до соседних, то мы в состоянии выполнить поиск по критерию стоимости. Ближайший пункт от Таллина – Хельсинки (100км), затем – Петербург (295), Рига (476), Тверь (716), Москва(161). Общая протяженность пути – 1748 км, и она существенно отличается от оптимальной – 957 км.

Предположим, что нам известно расстояние от каждого из городов до Москвы по прямой (рис.2.4).

Эта информация может использоваться для выбора следующего пункта маршрута. Так, доступными пунктами из Таллина являются Хельсинки (897 км), С.Петербург (634) и Рига (839). Стремясь на каждом шаге максимально приблизиться к цели, мы выберем на первом шаге С.Петербург, на втором шаге – Тверь, затем – Москву. Общая протяженность пути равна оптимальной.

Данный вид поиска называется **жадным поиском по первому наилучшему соответствию**. Каждый последующий узел для развертывания выбирается на основе **функции оценки f(n)**. В связи с тем, что мы не располагаем точными оценками, используются **эвристические функции h(n)** или эвристики. В рассмотренном примере эвристическая функция – это расстояние до цели по прямой.

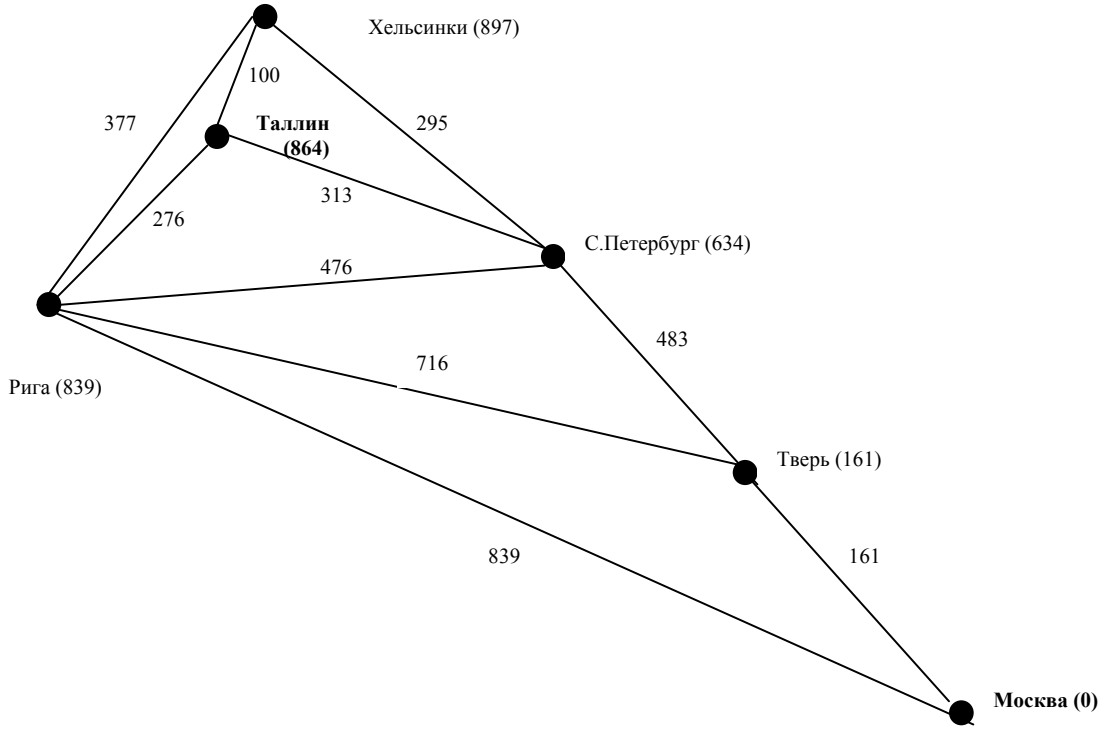


Рис.2.4. Расстояния до пункта назначения по прямой

Эта эвристика не является заведомо оптимальной. Если, например, между Тверью и Москвой самолеты не летают, то оптимальный маршрут Таллин – Рига – Москва будет существенно отличаться от предложенного алгоритмом жадного поиска по первому наилучшему совпадению: Таллин – С.Петербург – Тверь – Рига – Москва.

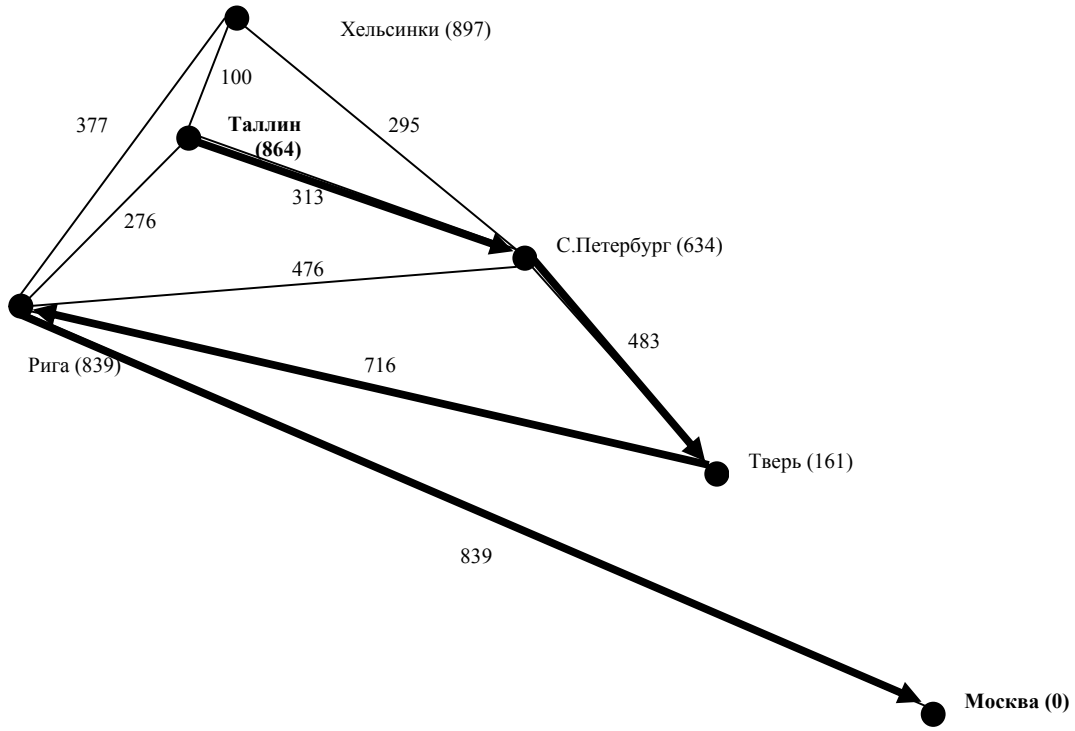


Рис.2.5. Жадный поиск по первому наилучшему соответствию

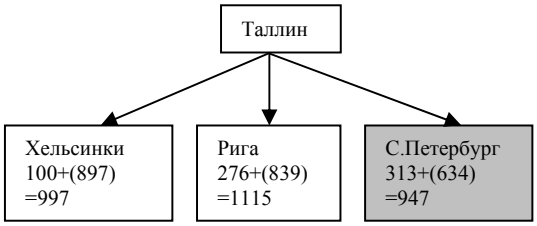
Жадный поиск по первому наилучшему совпадению напоминает поиск в глубину и страдает от тех же недостатков. На последней схеме без предотвращения повторяющихся состояний возможно бесконечное блуждание по отрезку С.Петербург – Тверь – С.Петербург и т.д. В наихудшем варианте сложность метода составляет b^m, где b – коэффициент ветвления, m – максимальная глубина пространства поиска. Однако, хорошая эвристическая функция позволяет существенно снизить такую сложность.

Оптимальный маршрут имеет минимальную протяженность, и это факт желательно использовать на каждом шаге поиска, а не в самом конце для оценки результата. Поскольку полными данными мы не располагаем, но у нас есть точная информация о пройденном пути и эвристическая оценка, мы можем использовать их для оценки общей протяженности пути на каждом этапе. Этот **метод минимизации суммарной оценки стоимости решения** называется **A* (А звездочка)**. Функция оценки f(n) на каждом шаге вычисляется как

$$f(n) = g(n) + h(n)$$

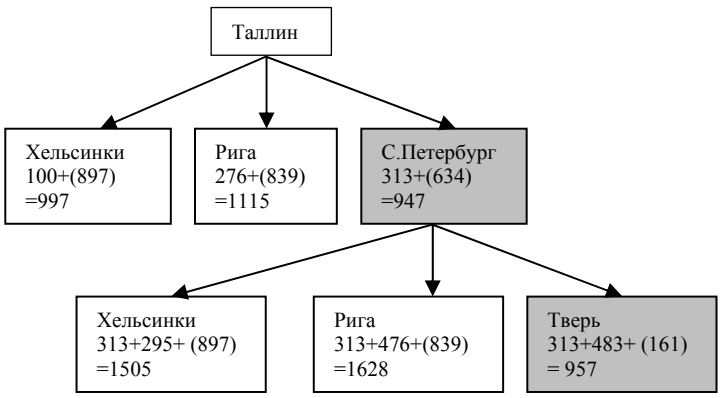
где g(n) – стоимость достижения n-го узла, h(n) – эвристическая функция стоимости достижения цели из n-го узла.

Рассмотрим функцию оценки для последнего варианта схемы воздушных трасс. При развертывании узлов от Таллина функция оценки будет принимать следующие значения.

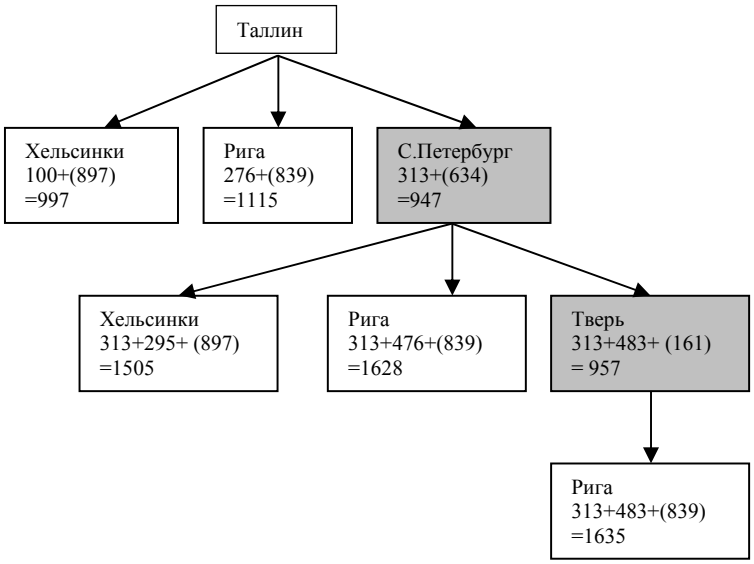


Следовательно, для развертывания узла будет выбран С.Петербург, как имеющий минимальное значение функции оценки.

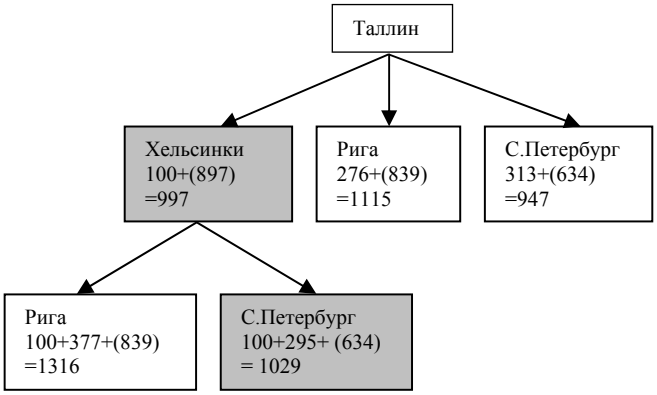
На втором шаге мы уже имеем накопленный путь 313 км, и функция оценки для развертывания узлов из С.Петербурга будет иметь следующие значения:



Следующим пунктом маршрута будет выбрана Тверь.

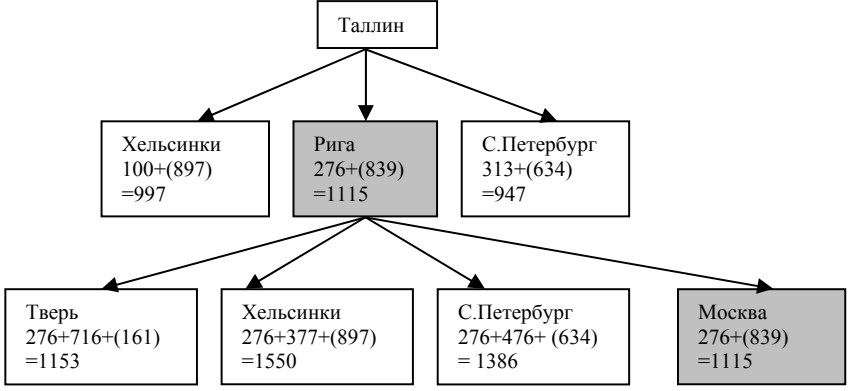


Из Твери есть путь только в Ригу, но функция оценки намного выше, чем при движении по маршруту Таллин – Рига. Следовательно, эта ветвь (Таллин – С.Петербург – Тверь – Рига – …) заведомо не дает оптимального решения, и для развертывания нужно вместо С.-Петербурга выбрать другой город. Следующим после С.Петербурга по возрастанию функции оценки является Хельсинки. При его развертывании получаем следующие значения функции оценки:



Здесь также имеем функции оценки для Риги и С.Петербурга выше, чем в ранее развернутых узлах. Следовательно, необходимо развернуть узел Рига:

Из Риги мы прямо попадаем в Москву. Решение найдено и оно является оптимальным. Заметим, что функция оценки является оптимистичной, поскольку ее часть – это расстояние по прямой, а реальный путь никак не может быть короче.



В задачах поиска важным является рациональный выбор эвристической функции. Рассмотрим еще один пример логической задачи. У нас она известна как игра «15». Мы же рассмотрим упрощенный вариант на девяти клетках с восемь фишками (игра «8»). Начальное и конечное состояния могут быть, например, следующими.

7	2	4
5		6
8	3	1

	1	2
3	4	5
6	7	8

Фишки можно двигать по горизонтали и вертикали на свободную клетку. Средняя сложность решения составляет 22 хода. Коэффициент ветвления – 3 (4, если свободная клетка в центре, 2, если – в углу). Число состояний примерно 3^{22} или $3.1*10^{10}$. Устраняя повторяющиеся состояния, можно сократить их количество примерно в 170 000 раз. Это состояние уже поддается контролю, но соответствующее количество состояний для игры в 15 равно 10^{13}, поэтому нужно найти хорошую эвристическую функцию. В качестве таких могут использоваться следующие функции:

h_1 – количество фишек, стоящих не на своем месте. На левом рисунке все фишки стоят не на своем месте, т.е. $h_1 = 8$. Данная эвристическая функция является допустимой, поскольку каждую фишку, стоящую не на своем месте, нужно переместить, по меньшей мере, один раз.

h_2 – сумма расстояний всех фишек от их целевых позиций. Функция также является допустимой, поскольку перемещение любой фишки на одну позицию меняет функцию на единицу. Данная функция более точно отражает степень расхождения текущей и целевой позиции. Расстояние измеряется в горизонтальных и вертикальных перемещениях. На левом рисунке $h_2 = 18$.

Вернемся к задаче о восьми ферзях. Комбинаторная сложность этой задачи диктует необходимость оценки каждой ситуации на шахматной доске, чтобы выбрать, как следует изменить расположение фигур. В качестве эвристической функции можно выбрать количество пар ферзей, которые атакуют друг друга прямо или косвенно (например, если все ферзи находятся на одной горизонтали, то реально бьют друг друга только соседние ферзи, а косвенно – каждый каждого). Используя такую эвристическую функцию, можно использовать алгоритмы **локального поиска**. Пусть первоначально

ферзи расставлены каждый на своей вертикали произвольно следующим образом:

18	12	14	13	13	12	14	14
14	16	13	15	12	14	12	16
14	12	18	13	15	12	14	14
15	14	14	♛	13	16	13	16
♛	14	17	15	♛	14	16	16
17	♛	16	18	15	♛	15	♛
18	14	♛	15	15	14	♛	16
14	14	13	17	12	14	12	18

В каждой свободной клетке показано значение эвристической функции при условии перемещения на эту клетку ферзя в пределах вертикали. Рамкой обведены клетки с минимальным значением эвристики. Таким образом, имеется 8 вариантов равноценных перемещений. После любого из них все эвристики пересчитываются и определяются кандидаты на следующее перемещение. Через пять ходов значение эвристики может достигнуть единицы, что очень близко к решению. Эта разновидность поиска называется **жадным локальным поиском**. Этот поиск часто показывает хорошую производительность, но и заходит в тупик по следующим причинам (рис.2.6):

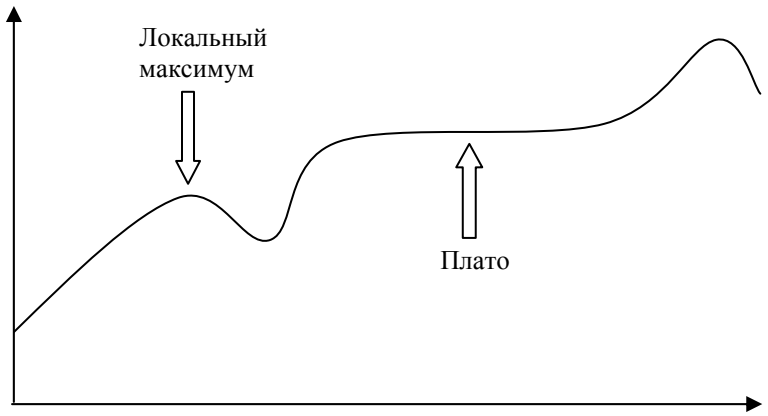


Рис. 2.6. Локальный максимум и плато

1. Локальные максимумы. Преодолеть их локальный поиск не в состоянии.
2. Плато. Область, в которой эвристика не меняется от хода к ходу.

Для устранения недостатков используются следующие модификации локального поиска: **движение в сторону** (разрешение на определенное число ходов при неизменной или ухудшающейся эвристике); **стохастический поиск с восхождением к вершине** (выбор случайным образом одного из вариантов восхождения к вершине); **поиск с восхождением к вершине** и перезапуском. Каждая из разновидностей поиска, естественно, требует увеличенного числа локальных итераций, но радикально ускоряет общее движение к цели. Так, поиск с восхождением к вершине и перезапуском для варианта с тремя миллионами ферзей позволяет находить решение меньше, чем за минуту.

Еще одной из разновидностей локального поиска является **генетический алгоритм**. Основными этапами алгоритма, обуславливающими его название, являются отбор, скрещивание и мутации. В задаче о восьми ферзях этот алгоритм может использоваться следующим образом:

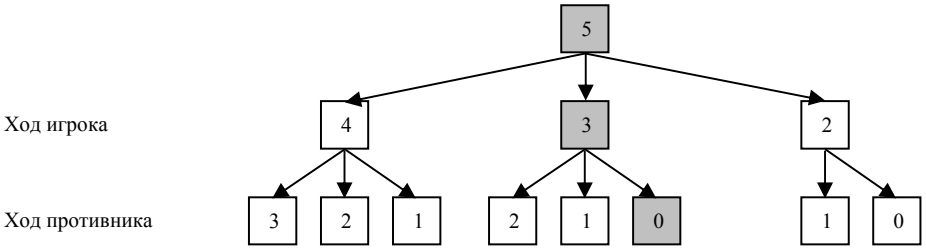
1. Выбираются позиции с лучшими значениями эвристик (отбор).
2. Доска «разрезается» по вертикали или по горизонтали, и части доски от разных позиций соединяются вместе (скрещивание).
3. В полученные новые комбинации вносятся случайные изменения (мутации).
4. Если полученная позиция хуже предыдущих, она отбрасывается, если лучше, запоминается (отбор).
5. Из имеющихся лучших позиций все повторяется с п.2.

Генетические алгоритмы широко используются при решении оптимизационных задач, хотя до сих пор не ясно, вызвана их популярность высокой эффективностью или эстетической привлекательностью и сходством с теорией эволюции Дарвина.

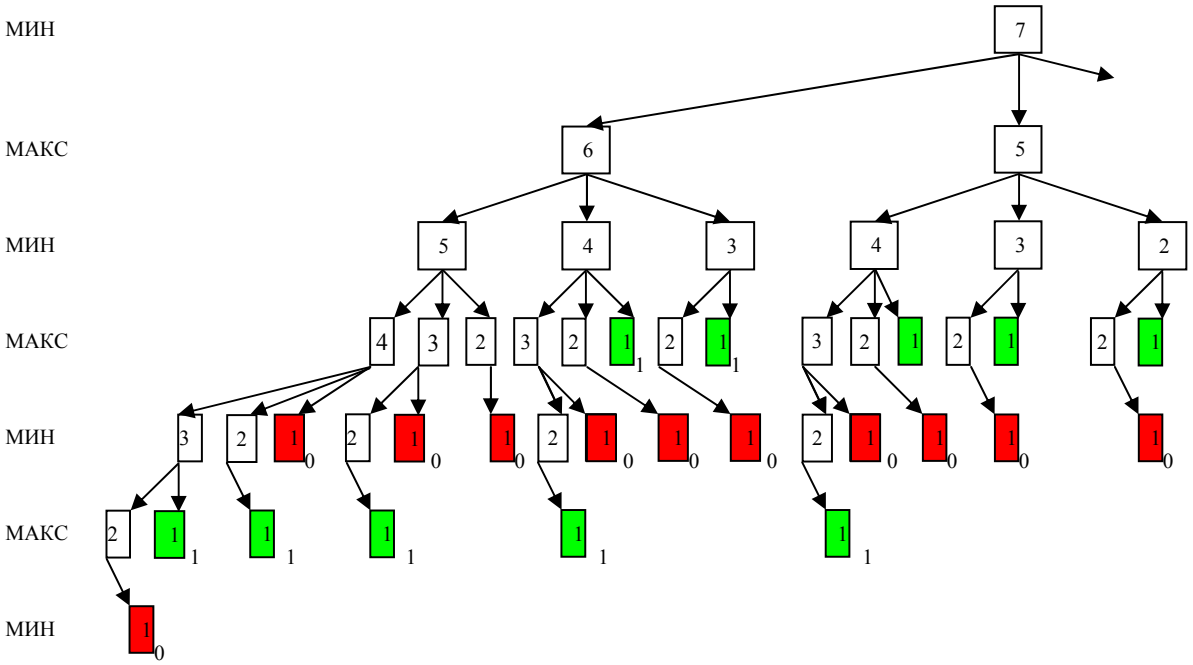
2.4. Поиск в условиях противодействия

В предыдущих задачах сложность нахождения решения определялась только размерностью пространства состояний. Существует класс задач, в которых присутствует элемент неопределенности. К ним относятся все игровые задачи. Мы не будем рассматривать шахматы (оцените сами комбинаторную сложность на первые 40 ходов, коэффициент ветвления на первом ходе 20), а вернемся к игре в спички.

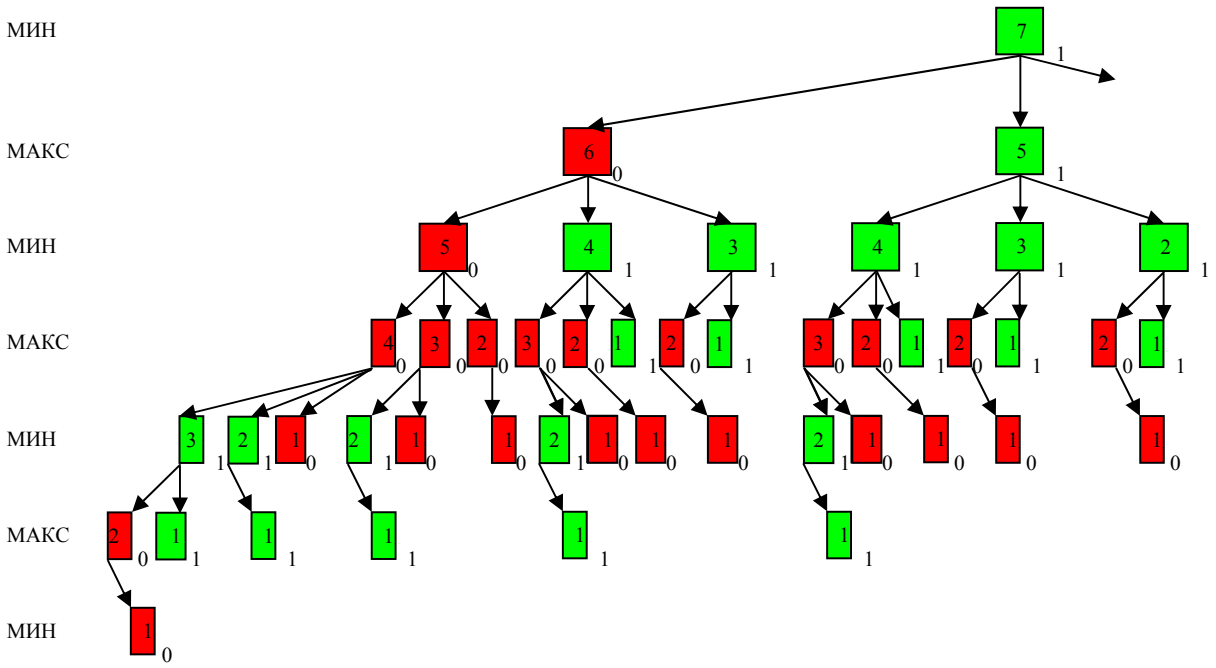
На первый взгляд здесь то же самое дерево решений. Игрок и его противник имеют 3 варианта хода на каждом этапе. Проблема заключается в том, что если мы выбрали ветвь дерева, которая приводит нас к победе, то это никак не устраивает противника, и он вовсе не будет двигаться по этой ветви, а примет все меры, чтобы не дать нам выиграть.



Так, показанная на рисунке цепочка 5 – 3 – 0 спичек нас устраивает (узлы отображают число спичек, остающееся после сделанного хода), но противник так никогда не будет ходить, а возьмет вместо трех спичек две, что приведет к нашему проигрышу. Чтобы оценить целесообразность того или иного хода, присвоим нашей победе значение 1, а победе противника – значение 0. В таком случае наша стратегия заключается в том, чтобы максимизировать результат, а стратегия противника – его минимизировать. Назовем для ясности противника – МИН, а себя – МАКС. Спускаясь по дереву решений, мы можем дать оценку каждому узлу на самом нижнем уровне (показан фрагмент дерева).



Выигрыш МАКСА (1) показан светлой заливкой, выигрыш МИНА (0) – темной. Предполагая, что оба игрока действуют в своих интересах разумно, мы можем дать оценку каждому ходу игроков на каждый уровень выше, а именно: из всех вариантов ходов МИН предпочтет те, которые дадут оценку 0, а МАКС – ходы, дающие оценку 1. Таким образом, оценка каждого хода МАКСА будет равна минимуму оценок ответных ходов МИНА и наоборот, оценка каждого хода МИНА будет равна максимуму оценок ответных ходов МАКСА.

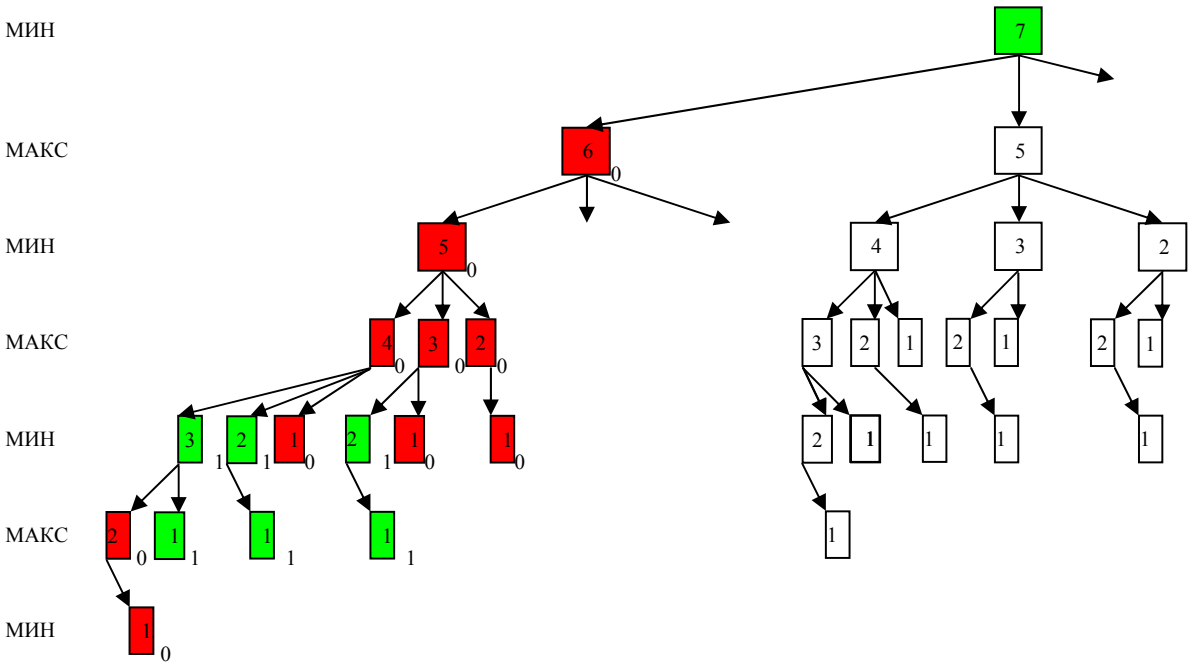


Данный алгоритм получил название **алгоритм минимакса или минимаксный алгоритм**. Легко видеть, что оценки по минимаксу являются пессимистическими для каждого из игроков и, следовательно, гарантируют успешный результат.

В общем случае минимаксный алгоритм состоит из следующих шагов:

1. Выбор шкалы оценок исходов игры.
2. Спуск по дереву и присвоение оценок конечным состояниям.
3. Последовательное присвоение оценок родительским узлам: для МИНА – максимальной из дочерних, для МАКСА – минимальной.
4. После присвоения значения начальной позиции можно начинать делать ходы.

Очевидным недостатком минимаксного алгоритма является его трудоемкость, поскольку необходимо выполнить обход всего дерева. Модификацией данного алгоритма, существенно укорачивающей его сложность является **альфа-бета отсечение**. Рассмотрим еще раз тот же фрагмент, теперь уже с позиции МАКСА, цель которого максимизировать результат. Спускаясь по дереву вглубь по левым ветвям, мы можем убедиться, что ход МАКСА 7 – 6 (взятие одной спички при семи в наличии) дает возможность МИНУ выиграть ходом 6 – 5. А поскольку оценка хода МАКСА определяется минимумом оценок возможных ответов МИНА, спускаться по соседним ветвям 6 – 4 и 6 – 3 нет необходимости и можно сразу переходить к спуску по ветви 7 – 5.



И наоборот, если мы видим, что ход МАКСА 7 – 5 имеет оценку 1, нам нет необходимости проверять ветвь 7 – 4.

Эффективность альфа-бета отсечения существенно зависит от порядка проверки узлов. Если бы мы вначале проверили ход 7 – 5, то нам бы не пришлось проверять ход 7 – 6. Это означает, что спуск по дереву целесообразно начинать с наиболее перспективных ходов. Разумеется, это возможно только если мы располагаем такой информацией.

2.5. Шахматные программы

Алгоритм минимакса даже при использовании альфа-бета отсечения все же требует спуска до терминального состояния по многим ветвям, следовательно, его применимость сильно ограничена. Иными словами, на практике он не используется, за исключением совсем простых случаев. Более практичным является применение эвристических оценок каждой позиции без спуска до нижних листов дерева. Такой подход используется, например, в шахматных программах, где шахматистами давно отработана методика оценки как отдельных фигур, так и позиций в целом. Так, пешка имеет стоимость 1, конь или слон – 3, ладья – 5, ферзь – 9. Оцениваются также такие характеристики, как безопасность короля, хорошая пешечная структура и т.д. Таким образом, каждый ход может быть сразу оценен. Это не значит, что можно ограничиться глубиной поиска в один ход. Хорошая позиция может быть достигнута через 5 или 8 ходов.

Модификация альфа-бета отсечения в этом случае заключается в том, чтобы ограничить верхнее значение оценки альфа (по принципу «от добра добра не ищут») и нижнее значение бета (минимально допустимое временное ухудшение позиции). Здесь все же всегда есть риск пропустить отличный ход или наоборот, «зевнуть». Более надежный подход заключается в использовании

ранее рассмотренного **итеративного углубления в пределах отведенного времени**. В этом случае в любой момент времени имеется все более совершенное решение, но выбор точки останова лежит вне программы, что нельзя признать удовлетворительным. Машина будет одинаково долго думать над простыми и сложными ходами. Одно из решений, называемое **поиском спокойных позиций**, заключается в том, что останавливать поиск можно только в спокойных позициях, когда от хода к ходу оценка позиции меняется незначительно. В позициях же, существенно меняющихся (например, проход пешки в ферзи), поиск надо продолжать.

При поиске спокойных позиций возникает проблема устранения **эффекта горизонта**, проявляющегося тогда, когда в перспективе имеется ход, причиняющий нам серьезный ущерб, который мы можем только отсрочить своими ходами, но являющийся неизбежным. Наши ходы, отвлекающие противника (например, объявление серии шахов), могут вывести опасный для нас ход за пределы **горизонта поиска**.

Первая шахматная программа была разработана в 1951 году Аланом Тьюрингом. Эта программа практически не эксплуатировалась, а ее алгоритм проверялся путем моделирования вручную.

Первой успешной отечественной программой стала **Kaissa**, разработанная в 1974 году в Институте теоретической и экспериментальной физики под руководством экс-чемпиона мира М.Ботвинника. Это программа победила на первом чемпионате мира по компьютерным шахматам в Стокгольме.

Наилучшей шахматной программой, которая победила Гарри Каспарова в 1997 году, является **Deep Blue**, созданная в компании IBM. Программа работала на параллельном компьютере с 30 процессорами IBM RS/6000. На этом компьютере эксплуатировались средства «программного поиска» и 480 специализированных СБИС шахматных процессоров, вырабатывающих ходы. На этом компьютере программа **Deep Blue** в среднем осуществляла поиск 126 миллионов узлов в секунду, а пиковая скорость составляла 330 миллионов. На каждый ход программа формировала до 30 миллиардов позиций, достигая глубины поиска 14. Основой программы является обычный альфа-бета поиск с итеративным углублением, но ключевой особенностью программы является способность углублять поиск в интересных позициях до 40 ходов. Помимо обычного поиска программа использовала справочник дебютов из 4000 позиций, большую базу эндшпилей и базу из 700 000 игр гроссмейстеров. Только такая добавка к программе позволила уравнять ее с чемпионом мира, который также обладает такими знаниями и использует шаблонные решения.

Группа разработчиков **Deep Blue** отказалась от предложенного Каспаровым реванша. Вместо этого на соревнованиях в 2002 году против Владимира Крамника выступила программа **Deep Fritz**, уже на обычном персональном компьютере. **Deep Fritz** – это разработка Франса Морха (Голландия) и Матиаса Фиеста (Германия). В этой программе применена техника нулевого хода (null move), заключающаяся в том, что в ходе поиска

игроку позволяется сделать два хода подряд (другой игрок пропускает ход). Благодаря этому легче обнаруживаются слабые ходы. Матч из восьми игр против **Deep Fritz** закончился ничьей, что позволило Крамнику заявить: «Теперь очевидно, что эта лучшая шахматная программа и чемпион мира играют на равных».

2.6. Контрольные вопросы

1. Какой вид поиска с каждой стороны должен использоваться при двунаправленном поиске?
2. Предложите эвристику для примера, рассматриваемого в подразд. 2.3, которая будет учитывать потери времени на промежуточных посадках и стыковках рейсов.
3. Оцените комбинаторную сложность игры в крестики-нолики на поле размером 3x3 и предложите метод сокращения размерности поиска. Насколько упрощается поиск?
4. Оцените комбинаторную сложность игры «23 спички» в случае развертывания дерева поиска от конечного состояния к начальному.

3. Поиск на основе логики

В разд. 2.1 мы рассматривали решение задачи о волке, козе и капусте методом поиска в пространстве состояний. Между тем, в литературе эта задача называется логической. Следовательно, ее решение должно быть получено путем умозаключений. Попробуем применить пропозиционную логику (булеву логику) для нахождения решения.

Обозначим состояния волка, козы, капусты и фермера переменными W, G, C и F соответственно и присвоим им значения «истина» или логическая единица, если они находятся на левом берегу, и «ложь» или 0, если на правом. Тогда стартовое состояние будет W=1, G=1, C=1, F=1, а конечное – W=0, G=0, C=0, F=0. Запрещенными состояниями будут следущие:

WG¬F – волк и коза на левом берегу, фермер – на правом;

GC¬F – коза и капуста на левом берегу, фермер – на правом;

¬W¬GF – волк и коза на правом берегу, фермер – на левом;

¬W¬GF – коза капуста на правом берегу, фермер – на левом.

Для решения этой задачи нужны входные и выходные переменные. Обозначим W1, G1, C1 и F1 в качестве выходных переменных. Решение задачи в логике первого порядка должно выглядеть следующим образом:

f(W,G,C,F) => (W1,G1,C1,F1) = (0,0,0,0).

К сожалению, эта задача не решается за один шаг, поэтому решение будет заключаться в последовательности преобразований переменных W,G,C,F в W1,G1,C1,F1.

При перемещении с левого на правый берег не должны быть истинными выражения W1G1 или G1C1.

(F & W & ¬G¬C) => (F1 = 0, W = 0, G1=G, C1=C) – фермер везет волка;

(F & G) => (F1 = 0, W1 = W, G1=0, C1=C) – фермер везет козу. Конфликта быть не может в принципе, так как волк не ест капусту;

(F & C & ¬W¬G) => (F1 = 0, C1 = 0, W1=W, G1=G) – фермер везет капусту.

Попробуем теперь выразить в виде булевой функции условие перемещения фермера с правого берега на левый без груза. Это возможно в том случае, если фермер находится на правом берегу (¬F), и с его уходом оттуда волк не съест козу (¬W¬G), а коза – капусту (¬G¬C):

(¬F & ¬W¬G & ¬G¬C) => (W1 = W, G1 = G, C1 = C, F1 = ¬F).

Таким же образом можно написать булевы функции для разрешения конфликтов на правом берегу. Объединив все эти импликации с помощью дизъюнкции, получим универсальную функцию для одного шага решения этой задачи, правда, без устранения повторяющихся состояний.

Заметим, что написание этих функций представляет собой весьма кропотливую работу, причем для другой задачи эти функции должны быть написаны заново. Формализация решения подобных задач может быть достигнута использованием таблиц истинности. Выпишем все возможные комбинации переменных:

№	W	G	C	F	W1	G1	C1	F1
1	1	1	1	1	1	1	1	1
2	1	1	1	1	1	1	1	0
3	1	1	1	1	1	1	0	1
4	1	1	1	1	1	1	0	0
5	1	1	1	1	1	0	1	1
6	1	1	1	1	1	0	1	0
...	…	…	…	…	…	…	…	...

Теперь удалим из таблицы строки, соответствующие запрещенным комбинациям. К таким относятся вышеперечисленные W1G1¬F1, G1C1¬F1, ¬W1¬G1F1, ¬W1¬G1F1, а также такие, в которых изменяется состояние более, чем одной из переменных W, G, C (по условию задачи на борт нельзя взять более одного груза), или не изменяется состояние переменной F (без лодки переправиться нельзя), либо лодка и груз движутся в противоположном направлении (одна из переменных меняется с 0 на 1, а F – с 1 на 0 и наоборот). Также из таблицы можно удалить строки, не отвечающие конечной цели (движение лодки порожняком с левого берега на правый). Таким образом, исходная таблица из 256 строк превращается в таблицу из 10 строк:

№	W	G	C	F	W1	G1	C1	F1
1	1	1	1	1	1	0	1	0
2	1	0	1	0	1	0	1	1
3	1	0	1	0	1	1	1	1
4	1	0	1	1	0	0	1	0
5	1	0	1	1	1	0	0	0
6	0	0	1	0	0	1	1	1
7	0	0	1	0	1	0	1	1
8	0	1	1	1	0	1	0	0
9	0	1	0	0	0	1	0	1
10	0	1	0	1	0	0	0	0

Поиск решения на основе таблицы истинности заключается в следующем:

1. Находим строку с исходным состоянием переменных (в нашем случае – первая строка).
2. полученные обновленные значения переменных W1, G1, C1 и F1 используем для поиска соответствующих значение переменных W, G, C и F в таблице.
3. Поиск заканчивается, когда значения W1, G1, C1 и F1 будут равны целевым значениям.

Данный метод позволяет сравнительно просто формализовать решение подобных задач. Проблема бесконечного блуждания, а в ее наличии легко убедиться, решается вычеркиванием в ходе поиска тех строк из таблицы, которые соответствуют уже посещенным состояниям.

Рассмотрим, каким образом можно применять логику первого порядка в задачах, где невозможен поиск методом проб и ошибок на примере игры

«Сапер». В начале игры мы не располагаем никакой информацией о расположении мин и первый ход делаем наугад. Если в открытой нами ячейке мины нет, то она открывается, а на смежных клетках отображается информация о количестве мин в восьми соседних клетках. Запишем это для ячейки (i,j) в виде $N_{i,j}$. Если это число равно нулю, то все соседние клетки открываются автоматически, поскольку тут решение совершенно очевидно. Если же $N_{i,j} > 0$, то требуется принятие решения, какую (какие) из соседних клеток пометить заминированными. Следует отметить, что, поставив флажок, мы лишь предполагаем, что там находится мина. Узнать этот факт точно мы можем только подорвавшись на ней. Тем не менее, мы будем считать такой факт установленным и будем обозначать его $M_{i,j} = 1$. Используя эти обозначения мы можем записать ситуацию после первого шага разминирования:

$M_{1,1} = 0$; $M_{1,2} = 0$; $M_{1,3} = 0$; $M_{1,4} = 0$;
$M_{2,1} = 0$; $M_{2,2} = 0$; $M_{2,3} = 0$; $M_{2,4} = 0$; $M_{3,1} = 0$;
$N_{1,4} = 1$;
$N_{2,1} = 1$; $N_{2,2} = 1$;
$N_{2,3} = 1$; $N_{2,4} = 1$; $N_{3,1} = 2$.

Исходя из имеющихся данных мы можем заключить (будем обозначать резолюции r1, r2 и т.д.):

r1: $M_{1,5} \vee M_{2,5}$
r2: $M_{1,5} \vee M_{2,5} \vee M_{3,3} \vee M_{3,4} \vee M_{3,5}$
r3: $M_{3,2}$
r4: $M_{3,2} \vee M_{3,3}$
r5: $M_{3,2} \vee M_{3,3} \vee M_{3,4}$
r6: $(M_{3,2} \,\&\, M_{4,2}) \vee (M_{4,1} \,\&\, M_{4,2}) \vee (M_{3,2} \vee M_{4,1})$

Можно заметить, что данные выражения некорректны с точки зрения булевой алгебры. Из r1 можно заключить, что мина находится либо в (1,5), либо в (2,5), либо в обеих клетках. В данной нотации используются т.н. **хорновские выражения**, где дизъюнкциями объединяются литералы, из которых один и только один является истинным. К данным резолюциям может быть применено **правило поглощения**:

$$A \,\&\, (A \vee B) = A$$

Применяя его к резолюциям r3, r4 и r5 получим $M_{3,2} = 1$, а значит, $M_{3,3} = 0$, $M_{3,4} = 0$.

Таким образом, мы можем сделать первый логический вывод о том, что в клетке (3,2) находится мина, а в клетках (3,3) и (3,4) их нет. Добавим эти факт в базу знаний о минном поле, отметим мину флажком и откроем клетки (3,3) и (3,4).

Напишем хорновские выражения для полученной ситуации:

r1: $M_{1,5} \vee M_{2,5}$
r2: $M_{1,5} \vee M_{2,5} \vee M_{3,5}$
r3: $M_{2,5} \vee M_{3,5} \vee M_{4,3} \vee M_{4,4} \vee M_{4,5}$
r4: $M_{4,2} \vee M_{4,1}$
r5: $M_{4,2} \vee M_{4,3} \vee M_{4,4}$

Применяя правило поглощения к выражениям r1, r2, r3, получим:

$(M_{1,5} \vee M_{2,5}) \ \& \ (M_{1,5} \vee M_{2,5} \vee M_{3,5}) = (M_{1,5} \vee M_{2,5})$,

следовательно, $M_{3,5} = 0$, и мы можем открыть клетку (3,5). После открытия клетки (3,5), а затем и (2,5), получаем следующую картину.

r1: $M_{1,6} \vee M_{2,6} \vee M_{3,6}$
r2: $M_{2,6} \vee M_{3,6} \vee M_{4,4} \vee M_{4,5} \vee M_{4,6}$
r3: $M_{4,3} \vee M_{4,4} \vee M_{4,5}$
r4: $M_{4,2} \vee M_{4,1}$
r5: $M_{4,2} \vee M_{4,3} \vee M_{4,4}$

Применить правило поглощения здесь невозможно, а значит, однозначный вывод о том, какую клетку открыть, сделать невозможно в силу неопределенности ситуации. Но это не значит, что мы не можем минимизировать риски, располагая данной информацией.

4. Вероятностные рассуждения

4.1. Нечеткая логика

Ранее мы рассматривали рассуждения на основе булевой логики, когда любая переменная либо истинна, либо ложна. Между тем, далеко не всегда мы располагаем полной информацией об истинности или ложности переменных, но имеем данные о степени их достоверности. Правила для вычисления достоверности сложных высказываний T следующие:

$T(A \wedge B) = \min(T(A), T(B))$

$T(A \vee B) = \max(T(A), T(B))$

$T(\neg A) = 1 - T(A)$

Продолжая рассмотрение игры «Сапер», будем считать, что мины разбросаны по полю равномерно. Тогда приведенные выше хорновские выражения r1, …, r5 означают, что мина находится достоверно в одной из клеток, т.е. вероятность нахождения мины в одной из клеток равна сумме равных вероятностей для каждой из клеток и равна единице:

r1: $p_{1,6} + p_{2,6} + p_{3,6} = 1$; $\quad p_{1,6} = p_{2,6} = p_{3,6} = 1/3$

r2: $p_{2,6} + p_{3,6} + p_{4,4} + p_{4,5} + p_{4,6} = 1$; $\quad p_{2,6} = p_{3,6} = p_{4,4} = p_{4,5} = p_{4,6} = 1/5$

r3: $p_{4,3} + p_{4,4} + p_{4,5} = 1$; $\quad p_{4,3} = p_{4,4} = p_{4,5} = 1/3$

r4: $p_{4,2} + p_{4,1} = 1$; $\quad p_{4,2} = p_{4,1} = 1/2$

r5: $p_{4,2} + p_{4,3} + p_{4,4} = 1$; $\quad p_{4,2} = p_{4,3} = p_{4,4} = 1/3$

Поскольку нашей задачей на данном этапе является не постановка флажка (это мы делаем только когда достоверно знаем, что там мина), а открытие клеток без мины, то нам удобнее рассматривать вероятности того, что в клетке мины нет. Обозначим соответствующие вероятности буквой q. Тогда

$q_{1,6} = q_{2,6} = q_{3,6} = 1 - 1/3 = 2/3$

$q_{2,6} = q_{3,6} = q_{4,4} = q_{4,5} = q_{4,6} = 1 - 1/5 = 4/5$

$q_{4,3} = q_{4,4} = q_{4,5} = 1 = 1/3 = 2/3$

$q_{4,2} = q_{4,1} = 1 - 1/2 = 1/2$

$q_{4,2} = q_{4,3} = q_{4,4} = 1 - 1/3 = 2/3$

В связи с тем, что резолюции r1,…,r5 объединены конъюнкцией, то вероятность отсутствия мины в каждой из клеток вычисляется как минимум вероятностей из каждой резолюции. Тогда получим итоговую вероятность для каждой клетки Q:

$Q_{1,6} = 2/3$; $\quad Q_{2,6} = 2/3$; $\quad Q_{3,6} = 2/3$;

$Q_{4,1} = 1/2$; $\quad Q_{4,2} = 1/2$; $\quad Q_{4,3} = 2/3$; $\quad Q_{4,4} = 2/3$; $\quad Q_{4,5} = 2/3$; $\quad Q_{4,6} = 4/5$.

Таким образом, не располагая достоверной информацией о том, где находятся мины, мы, тем не менее, можем утверждать, что выше всего вероятность отсутствия мины в клетке (4,6) – 4/5 против 1/2 и 2/3 в остальных рассматриваемых клетках – кандидатах. Отсюда следует не вполне очевидный вывод, что безопаснее открывать клетку (4,6).

Замечание. Если известно общее число мин, то этот факт может внести существенные коррективы в рассмотренную выше логику поиска. Если, например, известно, что число мин в 10 раз меньше общего числа клеток поля, т.е. вероятность нахождения мины в любой клетке, о которой нам ничего неизвестно, равна 1/10, что существенно ниже, чем в любой из рассмотренных клеток-кандидатов. Следовательно, следует открыть скорее совершенно неизвестную клетку, чем любую из соседних. Так, если мы открыли клетку в середине поля, и у нее есть один сосед с миной, то лучше поискать свободную от мин клетку где-нибудь в стороне, чем в соседних клетках, где вероятность напороться на мину равна 1/8.

4.2. Байесовские сети

Рассмотрим простой пример, близкий всем студентам. Чтобы сдать (Pass) экзамен, нужно подготовиться к нему (Study) или воспользоваться шпаргалкой (Cheat). Таким образом, имеется 3 булевых переменных. Хочется узнать вероятность успешно сдать экзамен.

В тех случаях, когда известны вероятности элементарных (атомарных) событий, можно воспользоваться методом вероятностного вывода на основе полного совместного распределения, которое описывается таблицей размерностью 2x2x2.

	Study		¬Study	
	Cheat	¬Cheat	Cheat	¬Cheat
Pass	0,15	0,4	0,04	0,06
¬Pass	0,01	0,04	0,05	0,25

Сумма всех вероятностей равна единице. В каждой клетке вероятность наступления элементарного события. Эта вероятность является результирующей, т.е. учитывает в себе все факторы. Так, вероятность успешной сдачи экзамена 0,4 учитывает вероятность подготовки к экзамену и вероятность того, что студент не воспользовался шпаргалкой.

Вероятности сложных событий легко вычисляются суммированием соответствующих строк или столбцов таблицы. Вероятность подготовки к экзамену равна сумме клеток левой половины таблицы и соответствует событиям (учил, не пользовался шпаргалкой и сдал; учил, пользовался и сдал; учил, пользовался и не сдал; учил, не пользовался и не сдал):

P(Study) = 0,15 + 0,4 + 0,01+ 0,04 = 0,6

Вероятность воспользоваться шпаргалкой равна сумме первого и третьего столбцов:

P(Cheat) = 0,15 + 0,01 + 0,04 + 0,05 = 0,25

Вероятность сдачи экзамена равна сумме клеток первой строки (учил, пользовался и сдал; учил, не пользовался и сдал; не учил, пользовался и сдал; не учил, не пользовался и сдал):

P(Pass) = 0,15 + 0,4 + 0,04 + 0,06 = 0,65

Метод вероятностного вывода на основе полного совместного распределения является скорее хорошей иллюстрацией принципа формирования вероятностей, чем практическим пособием, поскольку вероятности элементарных событий известны далеко не всегда. Чаще делают допущения о равной вероятности этих событий.

Обычно можно оценить вероятности истинности отдельных переменных. Пусть

P(Study) = 0,6 (в 40% случаев будут более важные дела, чем подготовка к экзамену), а

P(Cheat) = 0,25 (один шанс из четырех воспользоваться шпаргалкой).

Эти вероятности называются **априорными** или безусловными. Они представляют собой степень уверенности в истинности высказывания в отсутствии других данных.

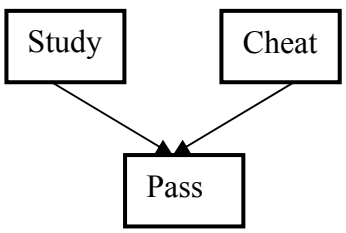
На первый взгляд, вероятность сдачи экзамена равна 0,6 + 0,25 = 0,85. На самом деле все сложнее. События Study и Cheat могут перекрываться, т.е. происходить совместно. Можно выучить материал и при этом для страховки воспользоваться шпаргалкой. Можно также все выучить, но не сдать (профессор придрался). Можно сдать без подготовки и шпаргалок (просто повезло).

Для нахождения искомой вероятности необходимо располагать условными вероятностями, например, P(Pass|Study) – вероятность сдачи экзамена при условии полной подготовки. В общем случае вероятность события A равна

$$P(A) = \sum_i P(A|B_i)\, P(B_i)$$

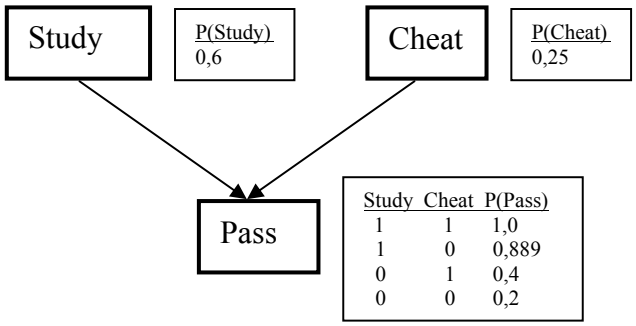
где $P(B_i)$ – априорная вероятность события B_i, $P(A|B_i)$ – условная вероятность события A при условии истинности события B_i. Условная вероятность связывает зависимые события. Если мы введем четвертую переменную – солнечную погоду (Sunny), то должны задавать условные вероятности типа P(A | Sunny). В реальных ситуациях мы можем столкнуться с тем, что размерность задачи будет из-за этого непомерно велика.

Для решения этой проблемы используются **байесовские сети**, которые позволяют установить зависимость переменных и упростить вычисление полного совместного распределения. Ниже приведена байесовская сеть для рассмотренного примера.



Обратим внимание, что в нашей модели переменные Study и Cheat являются независимыми. Возможна другая модель, когда пользование шпаргалкой обусловлено отсутствием подготовки к экзамену. Она будет рассмотрена позже.

Каждая вершина сети соответствует случайной переменной. Вершины соединяются направленными ребрами. Если стрелка направлена от A к B, то A называется родительской вершиной вершины B. Каждая вершина X_i характеризуется распределением условных вероятностей $P(X_i \mid Parents(X_i))$, которое количественно оценивает влияние на вершину ее родительских вершин. В нашем примере считаем известными следующие условные вероятности: вероятность сдать экзамен при условии подготовки и подстраховки шпаргалками равна единице; при подготовке и не пользовании шпаргалкой – 0,889; при условии пользования шпаргалкой без подготовки к экзамену – 0,4; а вероятность сдать экзамен без подготовки и шпаргалки – 0,2 (просто повезло).



Основной выигрыш при использовании байесовских сетей заключается в том, что вероятность любого состояния только на основе родительских (ближайших) вершин, а не всех вершин, от которых эта вершина зависит:

$$P(X_i \mid X_{i-1}, X_{i-2}, \ldots, X_1) = P(X_i \mid Parents(X_i))$$

В нашем примере мы имеем вершину Study, которая на самом деле может быть конечным результатом связанных событий: студент посещал лекции, имел конспект, у него был доступ к компьютеру, он располагал временем, и т.п. Как результат этих событий мы имеем факт подготовки к экзамену. Событие использование шпаргалки также является следствием целого ряда событий, вероятностями которых необходимо располагать: наличие времени, технических средств, подобающая одежда для скрытного использования шпаргалки. Для вычисления полного совместного распределения нужно знать все условные вероятности, например, вероятность наличия радиопередатчика при условии посещения лекций. Нам же для вычисления вероятности сдачи экзамена достаточно знать вероятности P(Study) и P(Cheat).

Приведенное выше правило называется **цепным правилом**. Следуя этому правилу достаточно просто вычислить вероятности событий, последовательно продвигаясь в направлении стрелок. В данном примере мы можем вычислить вероятность сдачи экзамена:

$$P(Pass) = P(Pass \mid Study, Cheat) * P(Study) * P(Cheat) +$$
$$+ P(Pass \mid Study, \neg Cheat) * P(Study) * P(\neg Cheat) +$$
$$+ P(Pass \mid \neg Study, Cheat) * P(\neg Study) * P(Cheat) +$$
$$+ P(Pass \mid \neg Study, \neg Cheat) * P(\neg Study) * P(\neg Cheat) =$$
$$= 1,0 * 0,6 * 0,25 + 0,889 * 0,6 * 0,75 + 0,4 * 0,4 * 0,25 + 0,2 * 0,4 * 0,75 = 0,65$$

В нашем сильно упрощенном примере этот выигрыш в сложности вычислений незаметен, поскольку цепочка байесовской сети не такая длинная. Второй фактор – независимость переменных Study и Cheat. Опытные студенты могут заметить некоторое неправдоподобие вероятностей: при условии

подготовки к экзамену шпаргалка лишь добавляет риск быть пойманным и выгнанным с экзамена. Изменим логику следующим образом. Будем считать, что пытаться воспользоваться шпаргалкой студент будет, только если не подготовится к экзамену. Байесовская сеть изменится так, как показано ниже.

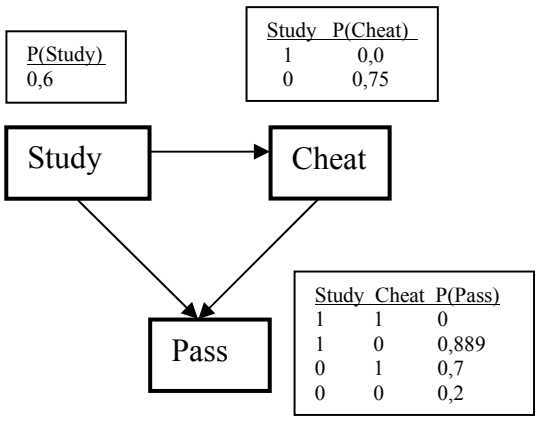
Условная вероятность воспользоваться шпаргалкой равна нулю в случае подготовки к экзамену и 0,75 при отсутствии подготовки. Вероятность

P(Pass | Study, Cheat) = 0.

Вычислим P(Cheat) и P(¬Cheat):

P(Cheat) = P(Cheat | ¬Study) * P(¬Study) = 0,75 *(1-0,6) = 0,3

P(¬Cheat) = 1-P(Cheat) = 0,7



Теперь, используя цепное правило, мы можем вычислить вероятность успешной сдачи экзамена:

P(Pass) = P(Pass|Study,¬Cheat) * P(Study) * P(¬Cheat) + P(Pass | ¬Study, Cheat) * P(¬Study) * P(Cheat) + P(Pass|¬Study, ¬Cheat) * P(¬Study) * P(¬Cheat)

= 0,889 * 0,6 * 0,7 + 0,7 * 0,4 * 0,3 + 0,2 * 0,4 * 0,7 = 0,513

Байесовские сети позволяют решать и обратные задачи. Например, известно, что студент экзамен сдал успешно. Требуется найти вероятность того, что он был к экзамену подготовлен. Для этого надо воспользоваться **правилом Байеса**:

P(A | B) = P(B | A) * P(A) / P(B)

В нашем случае вероятность сдачи экзамена, который был успешно сдан, P(Pass) = 0,513; вероятность сдачи экзамена при подготовке к нему P(Pass | Study, ¬Cheat) = 0,889; вероятность подготовки и не пользования шпаргалкой P(Study, ¬Cheat) = 0,6 * 0,7 = 0,42, следовательно,

P(Study, ¬Cheat | Pass) = P(Pass | Study, ¬Cheat) * P(Study, ¬Cheat) / P(Pass) =

= 0,889* 0,6 * 0,7 / 0,513 = 0,727

Найдем теперь вероятность того, что экзамен сдан с помощью шпаргалки:

P(¬Study, Cheat | Pass) = P(Pass | ¬Study, Cheat) * P(¬Study, Cheat) / P(Pass) =

= 0,7 * 0,4* 0,3 * / 0,513 = 0,164

И, наконец, вероятность того, что причиной сдачи экзамена было чистое везение:

P(¬Study,¬Cheat | Pass) = P(Pass | ¬Study,¬Cheat) * P(¬Study, ¬Cheat) / P(Pass)

= 0,2 * 0,4 * 0,7 / 0,513 = 0,109

Таким образом, байесовские сети обеспечивают декомпозицию сложных задач и при этом избавляют от необходимости задавать множество условных вероятностей.

4.3. Иллюстрация: Парадокс Монти Холл

Для иллюстрации применения правила Байеса рассмотрим следующую задачу, получившую в литературе название парадокса Монти Холл. Пусть Вы участвуете в телевизионной игре, в ходе которой Вам надо сделать выбор одной из трех дверей. За одной из них стоит автомобиль, за двумя другими – козы. Предположим, Вы указали на дверь №1. Ведущий открывает другую дверь, пусть это будет дверь №3, за которой обнаруживается коза, и предлагает Вам, пока не поздно, передумать и открыть дверь №2. Вопрос: имеет ли смысл прислушаться к мнению ведущего и открыть дверь номер два?

На первый взгляд, вероятности того, что машина находится за одной из оставшихся дверей равны $P(C_1) = P(C_2) = ½$, и менять решение бессмысленно. Однако, при внимательном подходе мы можем извлечь полезную информацию из факта открытии ведущим именно двери №3. Пусть априорные вероятности того, что машина находится за дверью №1, №2 и №3

$$P(C_1) = P(C_2) = P(C_3) = 1/3.$$

Найдем вероятности того, что ведущий откроет дверь №2 и №3 (дверь №1 он открыть не может, поскольку мы на нее уже указали). Если машина находится за дверью №1, то условные вероятности того, что он откроет двери №2 и №3:

$$P(D = 2 \mid C_1) = 0{,}5 \; ; \; P(D = 3 \mid C_1) = 0{,}5$$

То есть, ведущему все равно, какую из оставшихся дверей открыть. Если же машина находится за дверью №2, то вероятности того, что он откроет двери №2 и №3:

$$P(D = 2 \mid C_2) = 0 \; ; \; P(D = 3 \mid C_2) = 1$$

Поскольку машина стоит за второй дверью, то открыть эту дверь ведущий не может, а с вероятностью 1 откроет третью дверь. И наоборот, если машина стоит за третьей дверью, то соответствующие вероятности:

$$P(D = 2 \mid C_3) = 1 \; ; \; P(D = 3 \mid C_3) = 0$$

Итак, мы располагаем вероятностями того, какую дверь откроет ведущий в зависимости от того, где стоит автомобиль. Чтобы найти обратную условную вероятность, воспользуемся правилом Байеса. Поскольку известно, что D = 3, то

$$P(C_i \mid D = 3) = P(D = 3 \mid C_i) * P(C_i) / P(D=3)$$

Подставляя в эту формулу C_1 и C_2, получаем:

$P(C_1 \mid D = 3) = P(D = 3 \mid C_1) * P(C_1) / P(D=3) = ½ * 1/3 / 1/2 = 1/3.$

$P(C_2 \mid D = 3) = P(D = 3 \mid C_2) * P(C_2) / P(D=3) = 1 * 1/3 / 1/2 = 2/3.$

Таким образом, вероятность того, что машина находится за второй дверью, в два раза выше! Вывод, полученный «на кончике пера», может быть подкреплен и логическими рассуждениями: Если машина стоит за первой дверью, которую мы уже выбрали, то ведущему все равно, какую из оставшихся дверей выбрать. Если же машина за второй дверью, то у ведущего нет выбора, кроме двери №3. Сменив выбор, мы в худшем случае остаемся при тех же шансах, а в лучшем — получаем машину.

4.4. Обучение на основе наблюдений

Рассмотренные в предыдущем разделе примеры могут вызвать резонный вопрос: а как пользоваться этими замечательными инструментами в реальной жизни? Откуда взять все эти вероятности? Математики обходят это вопрос достаточно просто. Все рассуждения начинаются словами: пусть вероятность события A равна X. Таким образом, математики делают то, что можно могут, так, как нужно. Инженеру же приходится делать то, что нужно, так, как можно. Если не брать идеальные примеры, такие, как бросание игральных костей или раздача колоды карт, то найти вероятности можно только на основе наблюдений.

В нашем примере с экзаменом источником информации может стать только exit poll, т.е. анонимный опрос на выходе с экзамена. Результаты этого опроса могут выглядеть следующим образом:

№ п/п	Study	Cheat	Pass
1	Yes	No	Yes
2	No	Yes	Yes
3	Yes	No	Yes
4	Yes	No	No
5	No	Yes	No
6	No	No	No
7	Yes	No	Yes
8	No	Yes	No
9	Yes	No	Yes
10	No	No	Yes

Из этой выборки можно вычислить вероятность сдачи экзамена, которая равна отношению количества успешно сдавших к общему числу сдававших P(Pass) = 6 / 10 = 0,6. Аналогично получим вероятность подготовки P(Study) = 5 / 10 = 0,5 и пользования шпаргалкой P(Cheat) = 3 / 10 = 0,3. Таким же образом мы может получить условные вероятности P(Pass | Study) = 3 / 5 = 0,6 (отношение количества сдавших экзамен из тех, кто к нему готовился, к общему числу готовившихся к экзамену) и P(Pass | Cheat) = 1 / 3. Распределение удобно отобразить в виде дерева (рис.4.1):

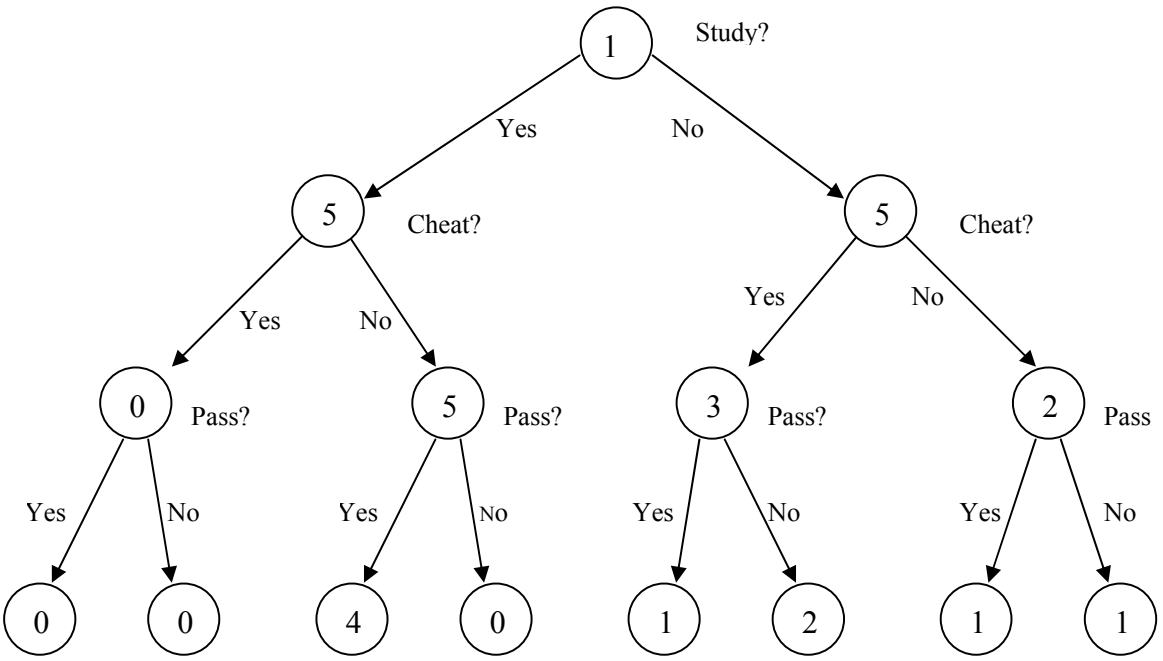


Рис.4.1. Дерево наблюдений

Здесь, как и в рассмотренных ранее задачах поиска, желательно сформировать дерево минимальной глубины. В этой связи необходимо вначале задействовать наиболее важные, т.е. наиболее информативные атрибуты поиска. Одним из подходящих критериев для выбора атрибута является объем информации, содержащийся в ответе.

Количество информации вычисляется по формуле Шеннона. Если возможные ответы v_i имеют вероятности $p(v_i)$, то информационное содержание фактического ответа I, измеренное в битах, равно

$$I(p(v_1), p(v_2), \dots, p(v_n)) = -\sum_i p(v_i) \log_2 p(v_i)$$

Вычислим количество информации в ответе о подготовке к экзамену:

$I(p(Study), p(\neg Study)) = - 5/10 * \log_2 5/10 - 5/10 * \log_2 5/10 = 1$ бит

Количество информации в ответе о шпаргалке:

$I(p(Cheat), p(\neg Cheat)) = - 3/10 * \log_2 3/10 - 7/10 * \log_2 7/10 = 1{,}533$ бит

Таким образом, ответ о шпаргалке в полтора раза более информативен, и в дереве поиска его следовало бы поставить раньше.

Другая проблема – достаточность информации в обучающей выборке. Приведенный выше пример дает весьма грубую оценку. Реальные вероятности могут существенно отличаться. Другая крайность – большой объем выборки, соизмеримый с генеральной совокупностью. Если мы устроим exit poll для всех студентов поголовно, то мы получим исчерпывающий ответ на все вопросы и в моделях уже нуждаться не будем. Полезность использования моделей достигается тогда, когда на небольшой выборке мы можем сделать вывод о всей генеральной совокупности. Например, узнав, что 10% студентов сдают экзамен с помощью шпаргалок, можно изменить формат экзамена, например,

разрешить пользоваться любыми источниками, но усложнить вопросы.

Одним из способов оценки достаточности выборки является постепенное добавление данных в первоначальную таблицу опроса со сравнением изменения вероятностей в каждой новой итерации. Как только изменение результатов станет меньше допустимой погрешности, обучение можно считать законченным.

5. Нейронные сети

5.1. Принцип построения нейронных сетей

Стандартный способ решения любой задачи с применением компьютера заключается в том, что задача исследуется, составляется алгоритм, который реализуется на языке программирования в виде программы. После отладки программа готова к эксплуатации.

Пусть нам поручено создать робот, играющий в баскетбол, а именно, та его часть, которая отвечает за бросание мяча в корзину. Первое, с чего нужно начать, это составить дифференциальные уравнения для полета мяча с разных позиций, внести поправки на сопротивление воздуха, параллакс датчиков по отношению к исполнительному устройству и т.д. После этого можно приступать к программированию. В то же время вряд ли кто-нибудь предположит, что Шакил О'Нил знает хотя бы элементарную математику, не говоря о высшей. Человеческий мозг, очевидно, решает эту задачу принципиально другим способом. Все знают, какой это способ – многократное повторение.

В предыдущем разделе мы рассматривали обучение, результатом которого являются вероятности наступления тех или иных событий. А возможно ли создать такую вычислительную структуру, которая бы без всякого программирования, на основе многократных повторений, могла бы обучаться решению задач. Такая структура называется нейронной сетью.

Идея позаимствована у природы. Нейронная сеть представляет собой совокупность нейронов – вычислительных элементов (иногда называемых персептронами), каждый из которых имеет несколько входов-синапсов и один выход-аксон.

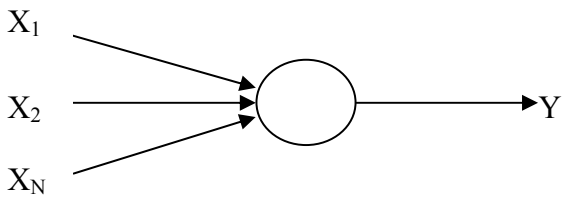


Рис.5.1. Одиночный нейрон

Интеллект одиночного нейрона (рис.5.1) невысок. Можно считать, что он реализует в сете простую регрессионную модель для N независимых переменных. Если же объединить множество нейронов, в сетевые структуры, то и реализуемая функция может быть сколь угодно сложной.

Изображенная на рисунке 5.2 сеть имеет явно выраженные слои, т.е. ряды нейронов, равноудаленные от входа (выхода). Могут создаваться и другие структуры, в том числе с обратными связями (рекуррентные).

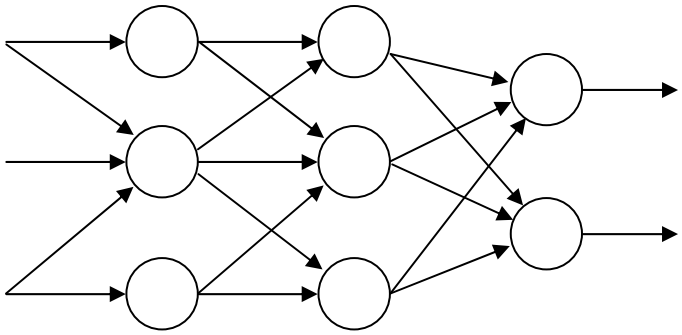


Рис.5.2. Нейронная сеть

Реализация такой сети может быть аппаратной, когда каждый нейрон выполнен на отдельном микропроцессоре, или программной, если нейроны эмулируются в специальных программах, подобных электронным таблицам.

Структура отдельного нейрона может быть произвольной, но чаще всего используется следующая (рис.5.3):

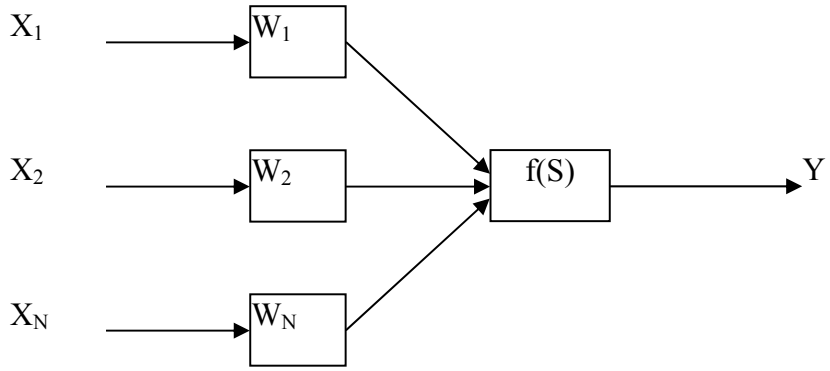


Рис.5.3. Структура нейрона

Входные сигналы (переменные) X_i взвешиваются (умножаются на коэффициенты W_i , называемые синаптическими весами), затем суммируются, и полученная взвешенная сумма

$$S = W_1X_1+W_2X_2+\ldots+W_NX_N$$

подвергается изменению функцией f(S), называемой функцией активации. Выходной сигнал Y также может повергаться взвешиванию (масштабированию). В качестве функции активации чаще всего используются сигмоидная функция $Y = 1 / (1 + \exp(-\lambda S))$, а также гиперболический тангенс, логарифмическая функция, линейная и другие. Основное требование к таким функциям – монотонность.

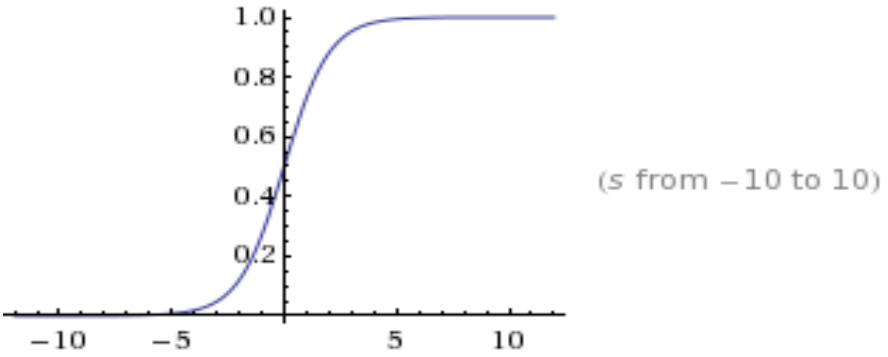


Рис.5.4. Функция логистического сигмоида

Рассмотрим теперь, каковы вычислительные возможности единичного нейрона. Пусть число синапсов равно трем, и синаптические веса W_1, W_2, W_3 равны 1.0 , а функция активации имеет следующий вид, представленный графиком (это функция логистического сигмоида, **смещенная вдоль оси X вправо** на 0,5 (рис.5.5):

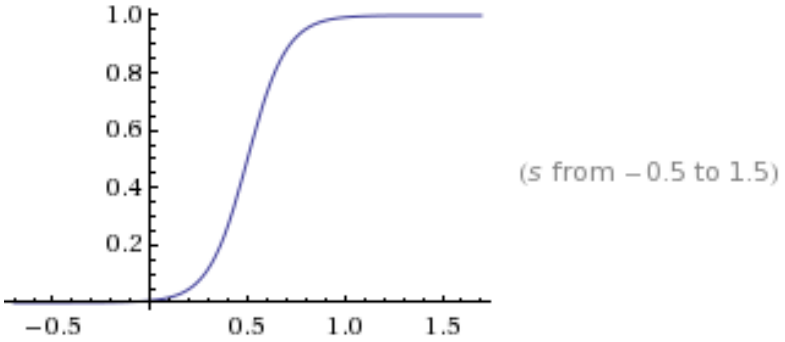


Рис.5.5. Вид функции активации для реализации булевых функций

Пусть входные сигналы принимают значения 0 или 1. В этом случае на выходе будет приблизительно 1 в случае, если хотя бы один из входных сигналов равен 1, то есть функцию дизъюнкции. Если же синаптические веса W_1, W_2, W_3 установить в 0.2, то этот же нейрон будет реализовывать функцию конъюнкции.

5.2. Обучение нейронной сети

Аналогия с мозгом не заканчивается на структуре нейрона и сети нейронов. Из природы позаимствована также идея обучения нейронных сетей. Известно, что человеческий мозг способен к самообучению, причем достигает успехов зачастую, не зная природы процессов, лежащих в основе выполняемых действий. Например, чтобы попасть мячом в баскетбольное кольцо, робот-баскетболист должен измерить расстояние до кольца и направление, рассчитать

параболическую траекторию, и совершить бросок с учетом массы мяча и сопротивления воздуха. Человек же обходится без этого только через тренировки. Многократно совершая броски и наблюдая результаты, он корректирует свои действия, постепенно совершенствуя свою технику. При этом в его мозгу формируются соответствующие структуры нейронов, отвечающие за технику бросков. Таким образом, непременным атрибутом обучения является многократное повторение и возможность немедленной оценки полученного результата.

Для нейронных сетей это процесс может быть представлен следующим алгоритмом (рис.5.6):

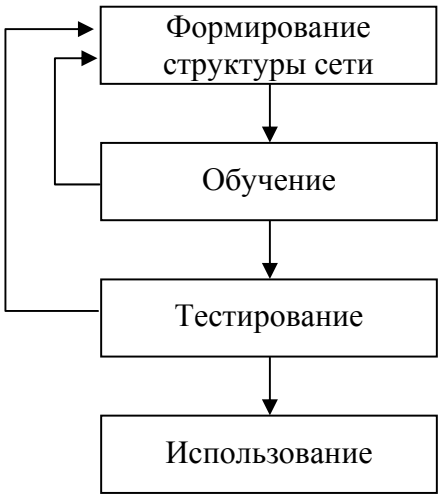


Рис.5.6. Процесс создания и использования нейронной сети

Выбор структуры нейросети представляет собой отдельную задачу и заключается в выборе топологии сети и функций активации каждого нейрона. Вначале параметры нейронов устанавливаются произвольно.

Обучение заключается в том, что на вход сети подаются специальные тренировочные данные, то есть такие входные данные, выходной результат для которых известен. На выходе формируются результирующие данные, результаты сравниваются с ожидаемыми, и вычисляется значение ошибки. После этого в определенной последовательности выполняется коррекция параметров нейросети с целью минимизации функции ошибки. Если удовлетворительной точности достигнуть не удается, следует изменить структуру сети и повторить обучение на множестве тренировочных данных.

После того, как сеть обучена, выполняется тестирование, то есть контроль точности на специальных тестовых данных. Это означает, что все данные следует разбить на два подмножества: на первом из них выполняется обучение сети, а на втором – тестирование. Это разбиение может быть случайным или регулярным, например, каждая вторая запись исходного массива данных может использоваться для тестирования. Тестирование от обучения отличается тем, что на тестовых данных только проверяется точность, а, поскольку эти данные

не используются для подбора параметров сети, они могут служить критерием качества обучения. По аналогии с обучением человека тестирование можно уподобить экзамену.

В качестве примера рассмотрим задачу оптического распознавания текстов. Пусть на вход сети в качестве входных сигналов подается матрица точек выделенного фрагмента изображения, соответствующая распознаваемому символу (рис.5.7). Пусть на вход сети подается матрица точек распознаваемого знака. На выходе формируются сигналы, соответствующие распознанному знаку. Обучение сети заключается в многократном «предъявлении» сети разных вариантов начертания символов вместе с готовыми «ответами».

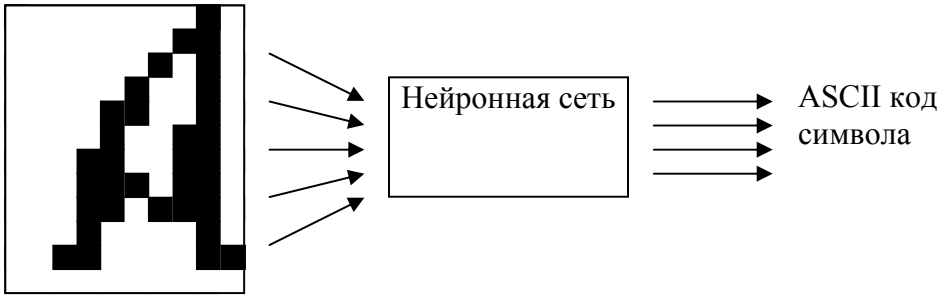


Рис.5.7. Нейронная сеть в задаче распознавания символов

Заметим, что сеть не пытается запомнить все возможные варианты начертания каждого символа, а лишь формирует выходной сигнал

$$Y_j = f(X_1, X_2, \ldots X_n)$$

как функцию от входных переменных. Заметим, что такой подход к распознаванию текстов обладает существенным преимуществом перед прочими в быстродействии. Так же и человек: чтобы прочитать текст, не спрашивает названия шрифта, и не ищет в памяти все возможные начертания символов. Ассоциации в мозгу устанавливаются мгновенно.

Задача обучения сети имеет огромную размерность. Так, для обучения сети, состоящей всего из 10 нейронов, в каждом из которых по 3 синапса, необходимо подобрать значения по меньшей мере 40 параметров (30 значений W_i – синаптических весов, и 10 параметров функций активации λ_i). Если каждый из параметров подбирать с дискретностью 1/100, то общее число прогонов сети на множестве тренировочных данных составит 100^{40}. Очевидно, что такая задача не под силу даже суперкомпьютерам.

Данная задача удовлетворительно решается с помощью алгоритма обратного распространения (*back propagation*), который заключается в следующем.

Вначале все параметры сети устанавливаются произвольно.

1. Через сеть прогоняются тренировочные данные, и вычисляется суммарная функция ошибки $E = sum(E_i^2)$, где $E_i = Y_i - y_i$, Y_i – вычисленное значений выходной величины, y_i – ожидаемое значение.

2. Вычисляется значение производных функции ошибки по каждому параметру, а на их основе – расчет поправок к параметрам нейронной сети.
3. Параметры сети корректируются на величину поправок, после чего шаги 2 и 3 повторяется с начала до тех пор, пока функция ошибки не снизится до заданного уровня.

Несмотря на простоту, данный алгоритм является весьма трудоемким, и его ускорение представляет собой актуальную задачу.

Если в результате обучения не был получен удовлетворительный результат, то необходимо изменить структуру сети. Это может быть сделано вручную, либо структура может выбираться из заранее созданного набора (библиотеки структур). Программные продукты, поддерживающие такие решения, существуют. Но наиболее удачным следует признать подход, при котором структура сети формируется автоматически. Примером может служить нейросеть NGO компании BioComp Systems Inc. (www.biocompsystems.com).

Данный подход заключается в применении к этой задаче генетических алгоритмов. Дело в том, что в процессе обучения сети выявляются «сильные» нейроны и связи между ними (чувствительные к изменению параметров) и «слабые» (параметры которых можно менять произвольно без существенного влияния на конечный результат). Используя эти данные, можно управлять популяцией нейронов. Слабые нейроны и синапсы должны отмирать, а для развития структуры, а также чтобы предотвратить всеобщее «вымирание», сеть подвергается «мутации»: в нее случайным образом или другим способом, например, для усиления «сильных» нейронов добавляются новые нейроны и синаптические связи. Таким образом, через множество поколений, количество которых может достигать десятков тысяч, сеть будет иметь оптимальную структуру.

В связи с этим может возникнуть вопрос: а зачем тратить такое количество времени на оптимизацию структуры сети, если обучение сети максимальной размерности с полным набором связей займет заведомо меньшее время? К тому же ранее было сказано, что быстродействие обученной нейронной сети достаточно велико.

Причины здесь две. Первая заключается в том, что зачастую обученная сеть в дальнейшем реализуется на другой платформе, в частности, на аппаратном уровне. Вторая причина – оптимизация часто приводит к существенному снижению количества входных переменных за счет исключения избыточных, не влияющих на конечный результат. Этот факт придает нейросетям качественно новое свойство: можно не заботиться о том, какие входные данные являются важными, а какие нет – в процессе обучения лишние будут отброшены. В статистике подобную функцию выполняют пошаговая линейная регрессия, дисперсионный и факторный анализ.

Например, мы пытаемся использовать нейросеть для предсказания курса доллара. В качестве исходных данных мы можем подставить поведение этого курса в течение нескольких лет (кстати, такой способ предсказания величин на

основе только их поведения в прошлом называется «технический анализ»), экономические индикаторы и индексы (Dow Jones, NASDAQ, потребительские цены), а также любые другие, вплоть до данных метеонаблюдений. Если на всех этих данных будет получена нейросеть, дающая хорошие прогнозы, то пользоваться ею будет довольно утомительно, а иногда и дорого. При этом выявление значимых параметров само по себе представляет ценный результат. Если окажется, что на курс доллара оказывает влияние температура воздуха, можно исследовать природу данного феномена. Мне могут возразить, что такую связь можно обнаружить, сравнивая графики температуры воздуха и курса доллара. Однако, прямое сравнение позволяет обнаружить, главным образом, линейную зависимость, к тому же здесь в работу включается интеллект человека.

5.3. Особенности использования нейронных сетей

Полная автоматизация выбора топологии и параметров нейросети, предоставляемая пакетами программ, подобным NGO, может создать иллюзию того, что можно совсем не управлять процессом моделирования с помощью нейронной сети. Однако, применение нейронных сетей, как и статистических методов анализа, является творческим процессом, требующим понимания принципов работы данного инструмента. Следствиями неправильного построения и обучения нейросети, в основном, являются обобщение или переобучение. **Обобщением** называется излишнее упрощение сети, при котором она не воспроизводит мелкие зависимости. Например, обобщение сети для прогнозирования погоды приведет к тому, что она будет выдавать среднюю температуру для данного сезона. Крайняя степень обобщения в этой задаче – среднегодовая температура. Естественно, пользы от такого прогноза будет немного. **Переобучение** – это другая крайность, при которой нейронная сеть имеет излишне сложную структуру и в процессе обучения слишком тщательно подгоняет результаты к желаемым. Как результат, на данных, не участвующих в обучении, сеть показывает худшие результаты. Признак переобучения – существенная разница погрешностей на этапе обучения и на этапе тестирования. Еще одна опасность слишком длительного обучения в сочетании с генетической модификацией топологии сети – обесценивание результатов тестирования. Если при однократной процедуре обучения данные тестирования заведомо не участвуют в процессе обучения, то при модификациях сети данные тестирования используются для отбраковки неудачных топологий и, тем самым, оказывают влияние на конечную сеть.

Существенный недостаток нейронных сетей — неспособность к экстраполяции. Иными словами, нейросеть, обученная в некотором диапазоне значений входных переменных, не в состоянии прогнозировать за пределами этого диапазона. На практике это обычно выражается в том, что за пределами области обучения выходная переменная не меняется, как показано на рис.5.8. Если сеть обучена в диапазоне $-0,5 < X < 0,5$, то за пределами этого диапазона

график выходной функции обычно лежит горизонтально.

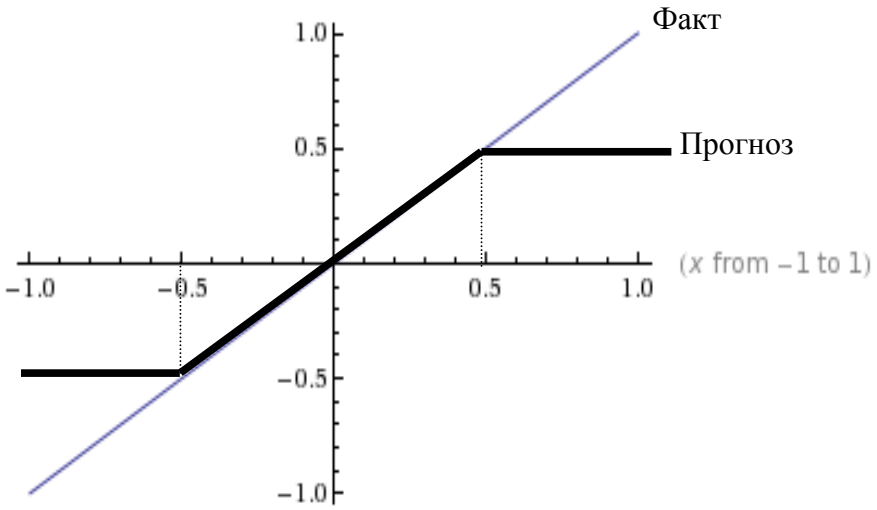


Рис. 5.8. Поведение нейронной сети вне области обучения

Несмотря на то, что нейронная сеть берет на себя любое, даже самое сложное преобразование входных переменных в выходные, предварительная обработка данных обычно не лишена смысла. Так, в случае прогнозирования на фондовом рынке, вероятно, целесообразным является переход от абсолютных значений котировок к относительным, т.е. к приращениям от одного наблюдения к другому. В таком случае для нейронной сети безразлично, колебания вокруг каких уровней использовать для обучения, если нас интересует лишь направление изменения котировок, вверх или вниз. В абсолютных же значениях сеть, обученная на колебаниях котировок вокруг значений 1000 или 5000 абсолютно бесполезна для прогнозирования в окрестности цифры 3000.

6. Экспертные системы

Экспертная система – это программа, которая в состоянии заменить собой человека-эксперта в его профессиональной деятельности. Структурно экспертная система состоит из базы знаний, машины вывода и пользовательского интерфейса.

База знаний имеет тот же смысл, что и в Прологе, т.е. состоит из фактов и правил. Машина вывода (**inference engine**) обращается к базе знаний (**knowledge base**) и преобразует запрос пользователя в ответ, задавая ему при необходимости вопросы, используя пользовательский интерфейс. Такая структура позволяет развивать экспертные системы, добавляя в нее новые знания, и при этом не требуется переписывать программу. Пустая экспертная система (без базы знаний) называется оболочкой (expert system shell) и может использоваться для многих предметных областей.

Создание экспертной системы заключается в формализации, т.е. преобразования знаний эксперта в форму, которая требуется для оболочки экспертной системы. Иными словами, требуется человек-эксперт, который является носителем знаний и в состоянии эти знания сформулировать для занесения в базу знаний. Этот факт является определяющим для выбора экспертной системы в качестве инструмента решения задачи. Человек-эксперт далеко не всегда в состоянии изложить свои знания в том виде, как этого требует формат базы знаний. В таких случаях вступает в действие инженер по знаниям (knowledge engineer), который является «переводчиком» между экспертом и базой знаний.

Рассмотрим работу экспертной системы на простом примере. Пусть наша база знаний предназначена для классификации животных. Предположим, что мы увидели животное и хотим ее идентифицировать. Пусть база знаний содержит всего двух животных, зебру и леопарда. Правила для описания этих животных (не в синтаксисе конкретной базы знаний, а в вольном переводе на русский язык) выглядят следующим образом:

«ЕСЛИ животное относится к классу млекопитающих И животное относится к виду хищников И животное имеет черные пятна, ТО животное – леопард»

«ЕСЛИ животное относится к классу млекопитающих И животное относится к виду травоядных И животное имеет черные и белые поперечные полосы, ТО животное – зебра»

Экспертная система начинает проверять гипотезы в порядке их расположения в базе знаний, в данном случае, начиная с леопарда. Чтобы установить истинность этой гипотезы, экспертная система должна сначала установить, относится ли животное к классу млекопитающих. Для этого она должна найти в базе знаний правило:

«ЕСЛИ женская особь животного имеет молочные железы, ТО животное относится к классу млекопитающих».

Теперь, чтобы установить принадлежность животного к млекопитающим, предстоит выяснить, имеет ли оно молочные железы. Если соответствующего правила в базе знаний нет, то пользователь должен сам дать ответ на этот вопрос. В этом случае экспертная система должна задать вопрос: **«ВЕРНО ЛИ, ЧТО женская особь животного имеет молочные железы?»**

Если пользователь ответит утвердительно, то факт «животное относится к классу млекопитающих» считается установленным (для простоты не будем рассматривать ситуацию, когда мы встретили самца). После этого правило для определения леопарда требует установить, является ли животное хищником. Установить это факт достоверно можно только будучи съеденным, поэтому, из гуманных соображений, в базу знаний помещено правило: **«ЕСЛИ животное имеет когти ИЛИ животное имеет клыки ТО животное относится к виду хищников»**

Пользователь не увидел ни клыков, ни когтей, а напротив, заметил копыта. Следовательно, гипотеза «леопард» отвергается, и экспертная система переходит к проверке следующей гипотезы «зебра». Факт «млекопитающее» уже установлен, и теперь нужно установить, относится ли животное к виду травоядных. Соответствующее правило выглядит следующим образом: **«ЕСЛИ животное имеет рога ИЛИ животное имеет копыта, ТО животное относится к виду травоядных».**

Экспертная система сначала задаст вопрос: «ВЕРНО ЛИ, ЧТО животное имеет рога?»

Пользователь отвечает отрицательно, а поскольку условия в данном правиле связаны операцией ИЛИ, задается следующий вопрос: «ВЕРНО ЛИ, ЧТО животное имеет копыта?»

Пользователь отвечает утвердительно, следовательно, факт «животное относится к виду травоядных» считается установленным. Для проверки гипотезы «зебра» остается проверить последнее условие: «ВЕРНО ЛИ, ЧТО животное имеет черные и белые поперечные полосы?»

Если полосы имеются, то гипотеза «зебра» подтверждается, а задача, поставленная перед экспертной системой, считается выполненной. Рис.6.1 содержит фрагмент простой экспертной системы, реализованной в среде VISIRULE, разработанной компанией LPA (http://lpa.co.uk).

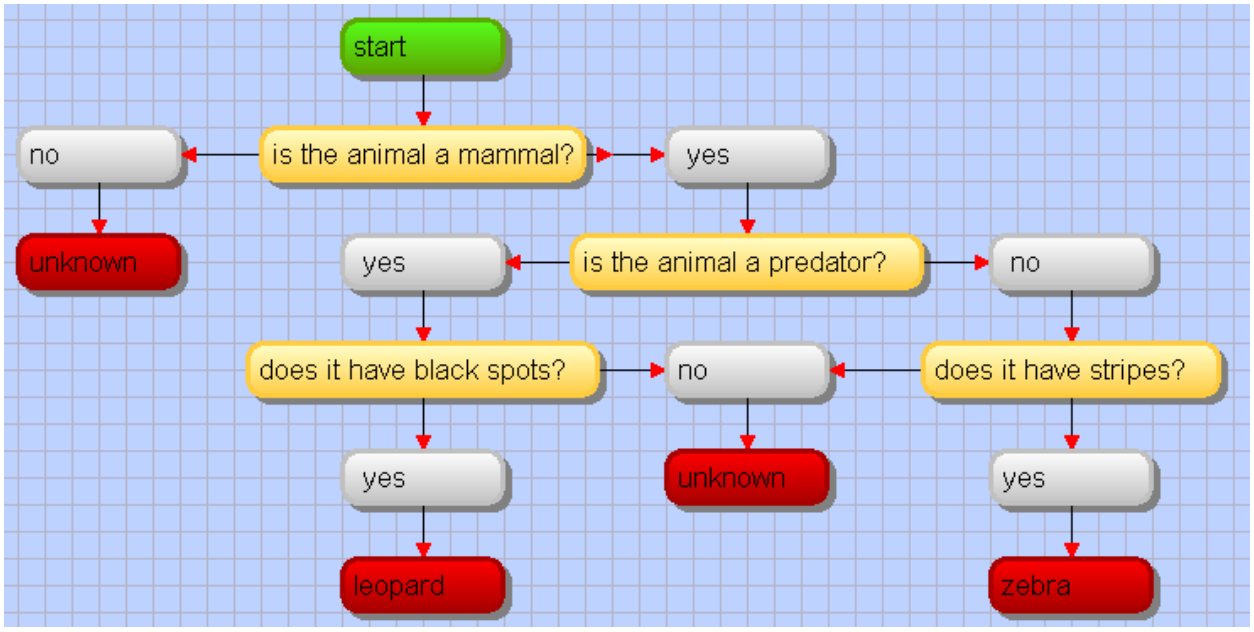


Рис.6.1. Пример представления правил

Основное требование к проектированию экспертных систем заключается в том, что пользователю должны задаваться такие вопросы, на которые он в состоянии ответить. Иными словами, экспертная система преобразует знания пользователя в знания эксперта. В приведенном выше примере идентификации животных явно неудачным вопросом является вопрос о молочных железах (особенно, если речь идет о случайно встреченном нами леопарде). Вместо него следует использовать правило:

«ЕСЛИ животное имеет волосяные покровы ТО животное относится к классу млекопитающих»

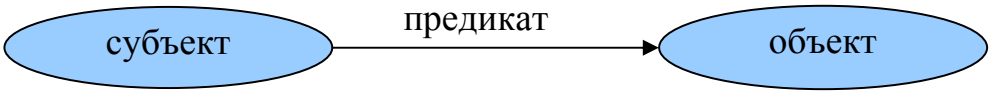
Другим важным свойством экспертной системы является способность объяснить, как был получен тот или другой вывод, что является лучшим подтверждением истинности полученного вывода. Если вспомнить рассказы А. Конан-Дойла, то нас, как и доктора Ватсона вывод вначале любой Шерлока Холмса кажется парадоксальным и выдуманным, но после объяснения, как этот вывод был сделан, мы удивляемся, что сами об этом не догадались.

Экспертные системы используются в самых различных областях знаний, в том числе в медицине (диагностика и лечение), геологоразведке (определение перспективности месторождений), химии (прогнозирование свойств органических соединений). В последнее время экспертные системы активно используются в Интернет-торговле для помощи покупателям в выборе товара. Наиболее известной является экспертная система ГУРУ на Яндексе (http://market.yandex.ru/guru-categories.xml), которая заменяет квалифицированного продавца-консультанта по самому широкому перечню товаров.

7. Семантические сети

7.1. Определение

Семантическая сеть - структура для представления знаний с помощью графа, в виде узлов, соединенных дугами. Узлы соответствуют понятиям, а дуги – отношениям между ними. Элементом семантической сети в простейшем случае является триплет следующего вида:



Одно и то же понятие может присутствовать в нескольких триплетах, что и обуславливает сетевую структуру. Основным свойством семантических отношений является **арность**, т.е. количество аргументов. Выше приведен пример **бинарного** отношения, т.е. отношения с арностью 2. Если все отношения в сети однотипные, то сеть называется **однородной**. Пример однородной сети – классификация биологических видов. В **неоднородной** сети количество типов отношений больше одного.

Основная цель применения семантических сетей для представления знаний – обеспечение независимости от языка, а также устранение неточностей и двусмысленностей, свойственных естественным языкам. Естественный язык – как живой организм, развивается и эволюционирует в сторону совершенствования основной своей цели – обеспечения понимания участников разговора. Однако, в силу природной лени носителей языка, эта оптимизация зачастую сводится к предельному упрощению конструкций, в результате чего смысл отдельной фразы выявить можно только из контекстного окружения. Хрестоматийный пример даже не двусмысленности, а «трехсмысленности» являет фраза «Он встретил ее на поляне с цветами». Совершенно непонятно, где цветы: у него, у нее или на поляне. От неоднозначности страдают все языки, включая английский. Так, вопрос “Tell me, what has four wheels and flies” заставляет нас вспомнить, какой летательный аппарат имеет четыре колеса. Речь при этом идет о “garbage truck”, и «flies» означает «мухи», а не «летает». Семантическая же сеть должна содержать знания в математически точной форме.

7.2. Историческая справка

Математика позволяет описать большинство явлений в окружающем мире в виде логических высказываний. Семантические сети возникли как попытка визуализации математических формул. Прародителями современных семантических сетей можно считать экзистенциальные графы, предложенные Чарльзом Сандерзом Пирсом в 1909г. Они использовались в органической химии для представления логических высказываний в виде особых диаграмм. Пирс назвал этот способ «логикой будущего».

Важным начинанием в исследовании сетей стали работы немецкого психолога Отто Зельца 1913 и 1922гг. В них для организации структур понятий и ассоциаций, а также изучения методов наследования свойств он использовал графы и семантические отношения. Научные изыскания Зельца имели огромное влияние на изучение тактики в шахматах, которые в свою очередь повлияли на таких теоретиков, как Саймон и Ньюэлл. Исследователи Дж. Андерсон (1973), Д. Норман (1975) и другие использовали эти работы для моделирования человеческой памяти и интеллектуальных свойств.

Что касается лингвистики, то первым ученым, занимавшимся разработкой графических описаний, стал Теньер. Он использовал графическую запись для своей грамматики зависимостей. Теньер существенно повлиял на развитие лингвистики в Европе. Компьютерные семантические сети были детально разработаны Ричардом Риченсом в 1956 году в рамках проекта Кембриджского центра изучения языка по машинному переводу. Процесс машинного перевода подразделяется на 2 части: перевод исходного текста в промежуточную форму представления, а затем эта промежуточная форма транслируется на нужный язык. Такой промежуточной формой как раз и были семантические сети. Первая такая система, которую создала Мастерман, включала в себя 100 примитивных концептов таких, как, например, НАРОД, ВЕЩЬ, ДЕЛАТЬ, БЫТЬ. С помощью этих концептов она описала словарь объемом 15000 единиц, в котором также имелся механизм переноса характеристик с гипертипа на подтип. Некоторые системы машинного перевода базировались на корреляционных сетях Цеккато, которые представляли собой набор 56 различных отношений, некоторые из которых - падежные отношения, отношения подтипа, члена, части и целого. Он использовал сети, состоящие из концептов и отношений для руководства действиями программы текстового разбора и разрешения неоднозначностей. Эти исследования были продолжены Робертом Симмонсом (1966), Уилксом (1972) и другими учёными [4].

В системах искусственного интеллекта семантические сети используются для ответа на различные вопросы, изучение процессов обучения, запоминания и рассуждений. В конце 70-х сети получили широкое распространение. В 80-х годах границы между сетями, фреймовыми структурами и линейными формами записи постепенно стирались. Выразительная сила больше не является решающим аргументом в пользу выбора сетей или линейных форм записи, поскольку идеи, записанные с помощью одной формы записи, могут быть легко переведены в другую. И наоборот, особо важную роль стали играть такие е факторы, как читаемость, эффективность, неискусственность и теоретическая элегантность, также учитываются легкость введения в компьютер, редак-тирование и распечатка [5].

На понимание проблем представления знаний с помощью семантических сетей большое влияние оказало эссе Дрю МакДермота «Искусственный интеллект сталкивается с естественной глупостью» [6], которое цитируется уже в течение 30 лет.

Наиболее продвинутой отечественной разработкой в области применения

аппарата семантических сетей в задачах анализа и поиска текстовой информации является набор программных продуктов под торговой маркой RCO (Russian Context Optimizer) компании «Гарант Парк Интернет» (www.rco.ru).

Наконец, следует упомянуть концепцию Семантической Паутины (Semantic Web), которая будет подробнее рассмотрена в подразд. 7.7 и 7.8. Семантическая Паутина – это дальнейшее развитие Всемирной паутины, заключающееся в том, что документы, размещаемые на ее ресурсах, содержат семантическую разметку, позволяющую извлечь смысл информации, содержащейся в этих документах. Такие семантические документы создаются, в основном, в формате RDF (Resource Description Framework) – расширении языка XML, дополненном средствами представления триплетов субъект-предикат-объект.

7.3. Типы семантических сетей

Семантическая сеть, в которой все отношения бинарные, образует **реляционный граф** (рис. 7.1.).

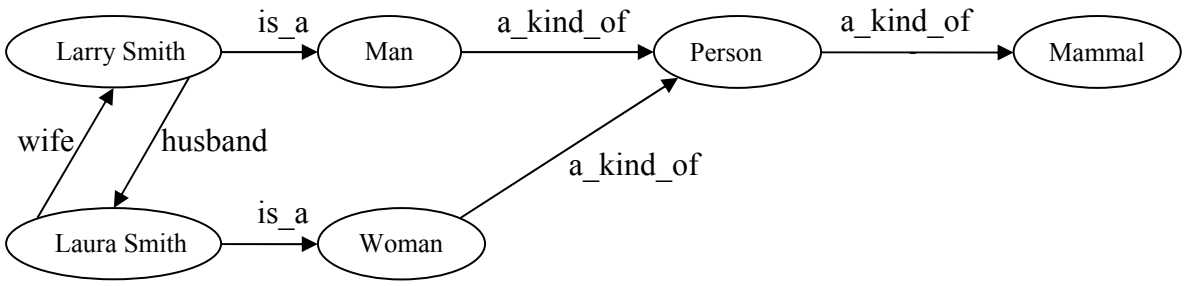


Рис.7.1. Пример реляционного графа

Несмотря на то, что понятия “Larry Smith” и “Man” обозначаются на графе одинаковыми эллипсами, их смысловое качество различно. “Larry Smith” – экземпляр класса мужчин, а “Man” – множество мужчин. В свою очередь, “Man” и “Woman” являются подмножествами более мощного множества людей (“Person”). Таким образом, как и в реляционных базах данных, здесь имеют место отношения «один к одному», «один ко многим», «многие к многим».

Если арность отлична от двух, изобразить семантическую сеть в виде графа уже сложнее. Но, как и в базах данных, можно провести нормализацию и привести все к бинарным отношениям. Например, унарное отношение типа «мотор – работает» можно привести к виду «мотор» -> «состояние» -> «работа». Отношение с арностью 3 типа «Самолет летит из Петербурга в Москву» оперирует с тремя понятиями: субъект – это самолет, а объекты – Москва и Петербург. Предикат – выполнение полета («летит»). Введя дополнительное понятие «полет», данное отношение можно представить

следующим фрагментом семантической сети (рис.7.2):

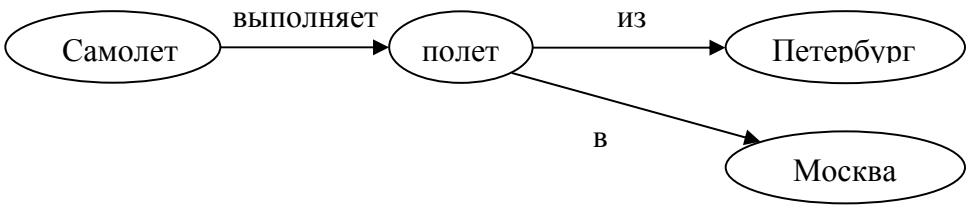


Рис. 7.2. К определению концептуального графа

Во многих случаях в качестве понятия в отношении участвует не простой объект или субъект, а другое отношение или целый фрагмент семантической сети. Например, мальчик видит в небе самолет и думает, что этот самолет летит из Петербурга в Москву.

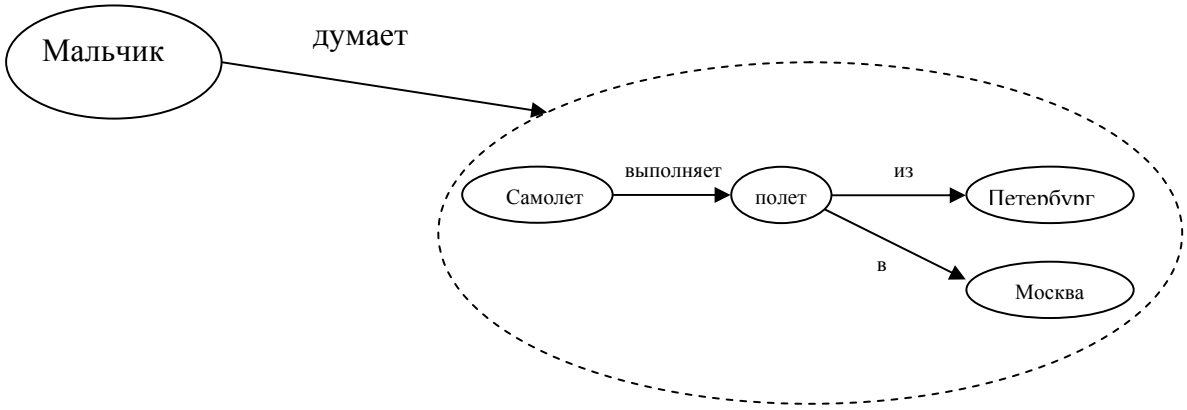


Рис.7.3. Концептуальный граф

Такие сети называются **пропозиционными сетями**, а граф такой сети – **концептуальным графом** (рис.7.3). Вложенность отношений в пропозиционных сетях может быть сколь угодно велика. Например, родители мальчика могут полагать, что он ошибается, если думает, что самолет летит из Петербурга в Москву, а их дедушка и бабушка, в свою очередь, могут считать, что родители недооценивают дедуктивные способности внука и т.д.

Для представления событий более подходит **граф с глаголом в центре** или **граф Растье**. Пример: Собака кусает почтальона (рис.7.4).

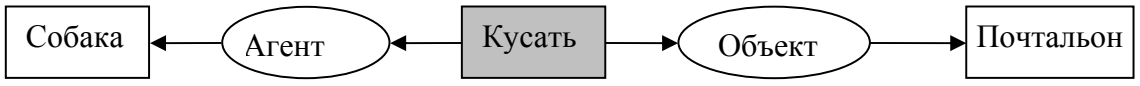


Рис.7.4. Граф Растье

В основе лежит не понятие, а действие или событие, в данном случае кусание. Субъектом или агентом кусания является собака, а объектом – почтальон. Граф с центром в глаголе позволяет обходиться без вложенности графов.

Мы можем достаточно просто расширить базу знаний о данном событии (рис.7.5):

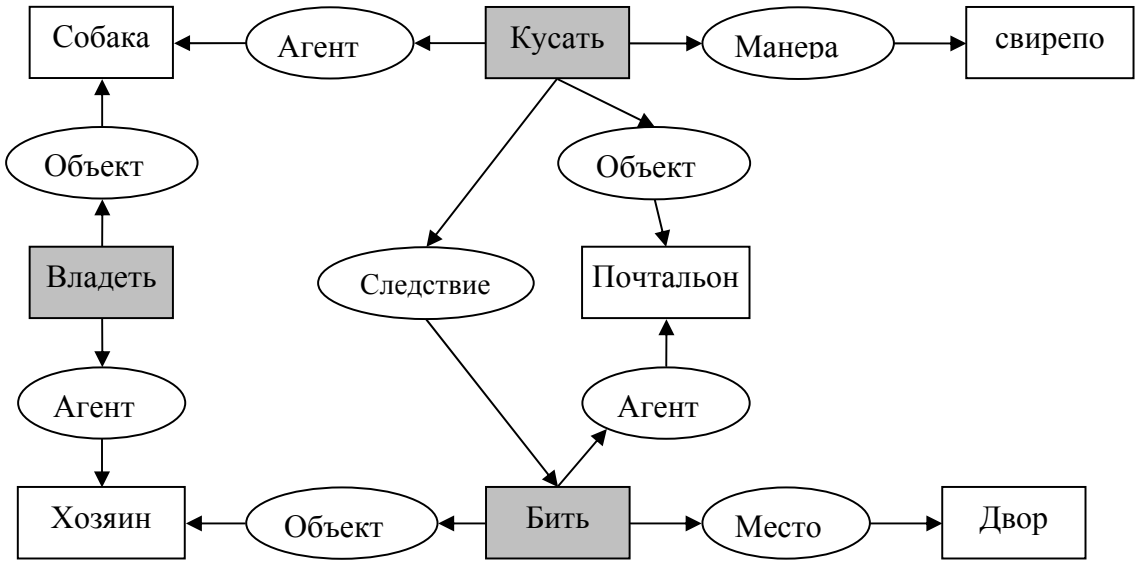


Рис.7.5. Семантическая сеть «Почтальон»

Следствием того, что собака свирепо покусала почтальона, стало то, что почтальон поколотил во дворе хозяина собаки.

7.4. Типы отношений в семантических сетях

Самым распространенным типом отношений в семантических сетях является иерархический тип, описывающий отношения между элементами, множествами и частями объектов. К ним относятся:

1) **отношение классификации ISA** (от английского “is a”). Говорят, что множество (класс) классифицирует свои экземпляры (например, “Сократ есть человек”). Иногда это отношение именуют “member of”. По-русски это может называться «есть» (единственное число) или «суть» (множественное число). Обратное отношение – “example of” или «пример».
2) **Отношение между множеством и подмножеством AKO** (“a kind of”), например, «Магистры подмножество студентов». Отличие от отношения ISA заключается в том, что классификация – отношение «один ко

многим», а подмножество – «много к многим». По-русски – «подмножество»

3) **Отношение целого и части**. Отношение меронимии – отношение целого к части (“**has part**”). Мероним – объект, являющийся частью другого объекта. Отношение холонимии – отношение части к целому (“is a part”). Рука – холоним для тела. Тело – мероним для руки.

Применяя иерархические типы отношений, следует четко различать, какие объекты являются классами, а какие – экземплярами классов. При этом вовсе не обязательно одно и то же понятие будет классом или экземпляром во всех предметных областях. Так, «человек» всегда будет классом в базах знаний типа «студенческая группа» или «трудовой коллектив», но может быть экземпляром класса млекопитающих в базе знаний по биологии.

Вершины семантического графа могут обозначать не только объекты, но и свойства или значения свойств. Отображение свойств на графе повышает его наглядность, но может сильно загромоздить его.

Кроме иерархических отношений в семантических сетях часто используются следующие типы отношений (во вторых скобках указаны типы вершин):

1) функциональные связи («производит», «влияет», …) (объект – объект);
2) количественные («больше», «меньше», «равно», …) (объект – объект или объект – свойство);
3) пространственные («далеко от», «близко к», «за», «над», «под», «выше», …) (объект – объект);
4) временные («раньше», «позже», «одновременно с», …) (объект – объект);
5) атрибутивные («иметь свойство», «иметь значение»,…) (объект – свойство или свойство - значение);
6) логические («и», «или», «не») (объект – объект или свойство – свойство);
7) лингвистические.

Число типов отношений может быть очень большим. Основная проблема при этом заключается в возможности идентификации этих отношений в запросах к базе знаний. В этой связи предпочтительным является сокращение числа типов связей (и вершин) за счет увеличения числа вершин.

Например, вместо отношения «семантсеть» - «предназначена» - «представление_данных» можно использовать «семантсеть» - «имеет» - «назначение»; «назначение» - «есть» - «представление»; «представление» - «чего» - «данных».

Сети с глаголом в центре (сети Растье) оперируют со следующими типами связей [5], приведенными в таблице 7.1. Рис. 7.6. содержит граф Растье, описывающий пример с почтальоном, где отношения приведены к стандартной форме.

Таблица 7.1. Типы связей в графах Растье

Имя	Тип	Определение	Упрощенное имя
(ACC)	accusative	Объект воздействия	PATient
(ASS)	assumptive	Точка зрения	PERspective
(ATT)	attributive	Свойство, характеристика	CHARacteristic
(BEN)	benefactive	Сущность, выступающая в роли выгодоприобретателя	BENeficiary
(CLAS)	classitive	Экземпляр класса	CLASsitive
(COMP)	comparative	Элементы, объединяемые сравнением	COMParison
(DAT)	dative	Получатель	RECeiver
(ERG)	ergative	Эргатив, агент процесса или действия	AGEnt
(FIN)	final	Результат или ожидаемая цель	GOAL
(INST)	instrumental	Использованные средства	MEAns
(LOC S)	spatial locative	Положение (позиция) в пространстве	SPAce
(LOC T)	temporal locative	Положение (позиция) во времени	TIME
(MAL)	malefactive	Сторона, пострадавшая в результате действия	MALeficiary
(PART)	partitive	Часть целого	PARTitive
(RES)	resultative	Результат, эффект, следствие	EFFect (или CAUse)

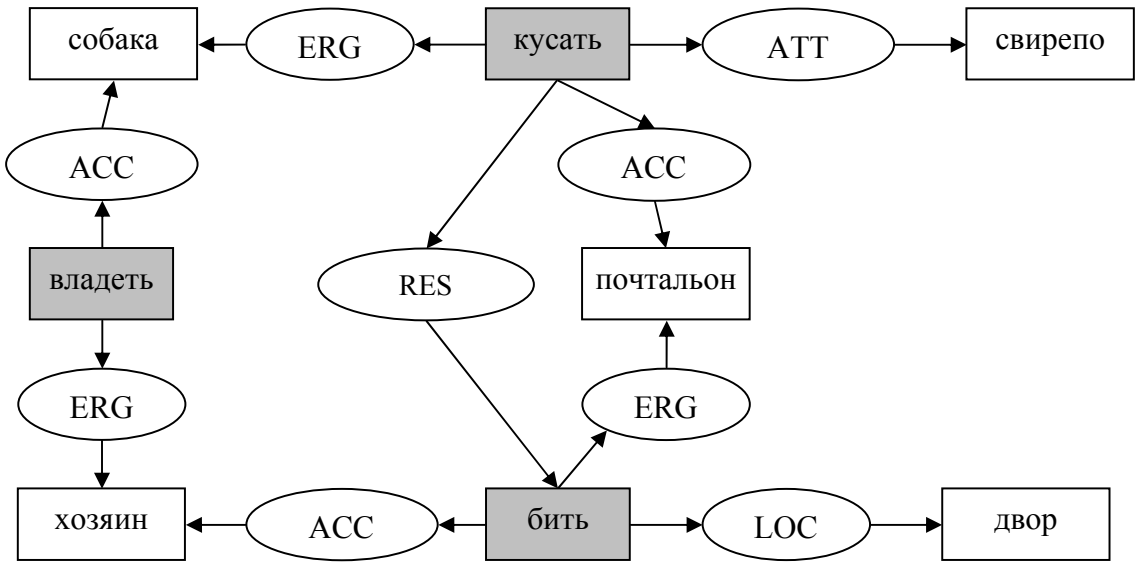


Рис.7.6. Граф Растье со стандартными отношениями

7.5. Онтологии и правила наследования отношений

Представление знаний – весьма сложный и творческий процесс. Основная проблема здесь заключается в том, что создание базы знаний почти всегда начинается «с нуля»; при этом отсутствуют т.н. знания начального уровня, которыми человек обзаводится, начиная с раннего детства. Создание таких баз бытовых знаний ведется уже более 25 лет компанией Cycorp. (www.cyc.com), однако, по признанию ее основателя Дагласа Лената (Douglas Lenat), машина все еще нуждается в том, чтобы ей в явном виде сообщали, что родители старше своих детей, и что люди перестают выписывать газеты, когда умирают. Знания конкретной предметной области можно разделить на общие для всех экземпляров и индивидуальные для каждого. Очевидно, что если формализация знаний для класса объектов будет выполнена один раз, то затем она может использоваться другими авторами при описании отдельных экземпляров.

Таким образом, формализация знаний любой предметной области должна базироваться на знаниях более общего уровня, описывающих основные отношения между объектами и общие свойства объектов. Такие знания называются онтологиями. Например, онтология для описания класса «человек», может содержать отношения “has part” с объектами «рука», «голова» и свойства «имеет имя», «имеет дату рождения» и т.п.

Применение таких знаний к объектам нижнего уровня осуществляется с помощью правил наследования:

Если X AKO Y и Y AKO Z то X AKO Z – если X является подмножеством Y, а тот, в свою очередь, частью еще большего множества Z, то класс X является подмножеством Z.

Если X ISA Y и Y AKO Z то X ISA Z – если X является экземпляром класса Y, а тот, в свою очередь, частью множества Z, то X является экземпляром класса Z.

Если X has_part Y и Y has_part Z то X has_part Z – если X имеет в своем составе Y, а тот, в свою очередь, имеет составную часть Z, то X имеет в своем составе Z.

Если X ISA Y и Y has_part Z то X has_part Z – если X является экземпляром класса Y, а тот, в свою очередь, имеет составную часть Z, то экземпляр X имеет в своем составе Z.

Если X AKO Y и Y has_part Z то X has_part Z – если X является подмножеством Y, а тот, в свою очередь, имеет составную часть Z, то класс X имеет в своем составе Z.

Если X ISA Y и Y has_a Z то X has_a Z – если X является экземпляром класса Y, а тот, в свою очередь, имеет свойство Z, то экземпляр X имеет свойство Z.

Если X AKO Y и Y has_a Z то X has_a Z – если X является подмножеством Y, а тот, в свою очередь, имеет свойство Z, то класс X имеет

свойство Z.

Использование правил наследования позволяет описания всех свойств каждого экземпляра указать лишь принадлежность к классу, для которого все необходимые свойства уже описаны.

7.6. Примеры

На рис.7.7 приведен граф семантической сети, воспроизводящий в себе известное стихотворение. Несмотря на исходную рекурсивность конструкции (существует даже программа на языке Prolog, генерирующая текст этого стихотворения), здесь не потребовалась пропозиционная сеть, поскольку все связи укладываются в обычные триплеты: старушка доит корову, а не процесс лягания коровой пса и т.д.

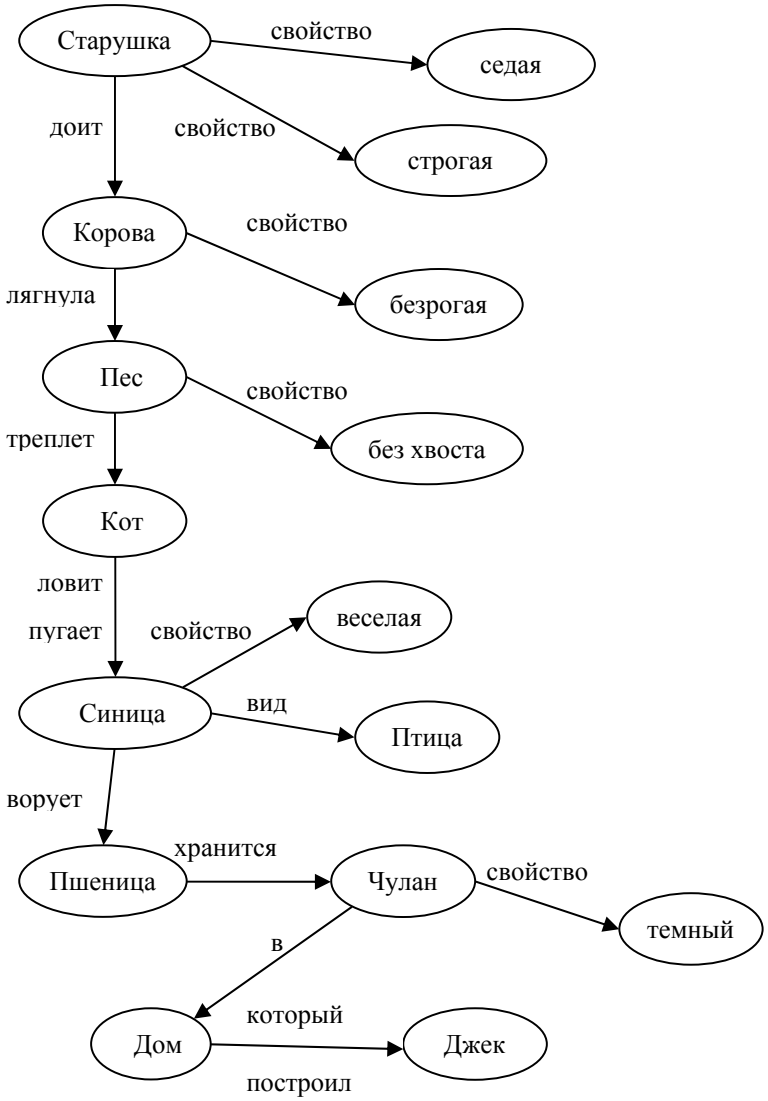


Рис.7.7. Граф семантической сети «Дом, который построил Джек»

Читатель, хорошо знакомый с оригиналом, может отметить, что граф не отражает тот факт, что синица ворует пшеницу *ловко*, а пес треплет кота *за шиворот*. Эти факты относятся не к объектам и субъектам, а к процессам или действиям. Синица может быть в целом неуклюжей, ловко только воровать пшеницу, причем не любую, а только ту, «которая в темном чулане хранится…». Ну а «за шиворот» - это не свойство пса или кота, а характеристика процесса трепания.

Второе свойство данного графа – статичность. Из графа, как, впрочем, и из текста не следует, что сначала кот ловит синицу, затем его треплет пес. Все эти действия – постоянные во времени, за исключением коровы, которая совершила однократный акт лягания пса, причем безотносительно его манипуляций с котом.

Другой пример перевода стихотворного произведения в семантическую сеть более схематичен и всего лишь отражает основной смысл. Для разнообразия возьмем англоязычный вариант (не будем разбираться, кто у кого списал).

"The Cicada and the Ant"
Jean de La Fontaine (1988)

Cicada, having sung her song
All summer long,
Found herself without a crumb
When winter winds did come.
Not a scrap was there to find
Of fly or earthworm, any kind.
Hungry she ran off to cry
To neighbor Ant, and specify:
Asking for a loan of grist,
A seed or two so she'd subsist
Just until the coming spring.

She said, "I'll pay you everything
Before fall, my word as animal,
Interest and principal."
Well, no hasty lender is the Ant;
It's her finest virtue by a lot.
"And what did you do when it was hot?"
She then asked this mendicant.
"To all comers, night and day,
I sang. I hope you don't mind."
"You sang? Why, my joy is unconfined.
Now dance the winter away."

Граф на рис.7.8 не воспроизводит художественную сторону произведения, а всего лишь отражает процессы, события и причинно-следственные связи между ними. Так, мы видим, что имеется процесс пения (“Sing”). Агентом (эргативом) пения является цикада, а позицией во времени – лето. Следствием пения цикады летом стало отсутствие пищи (“No food”) и голодание (“Hunger”) зимой (позиция во времени – “Winter”).Объект голодания (“accusative”) – цикада.

К сожалению, на графе лето и зима находятся в диаметрально противоположных точках, и факт наступления зимы после лета отразить не удалось.

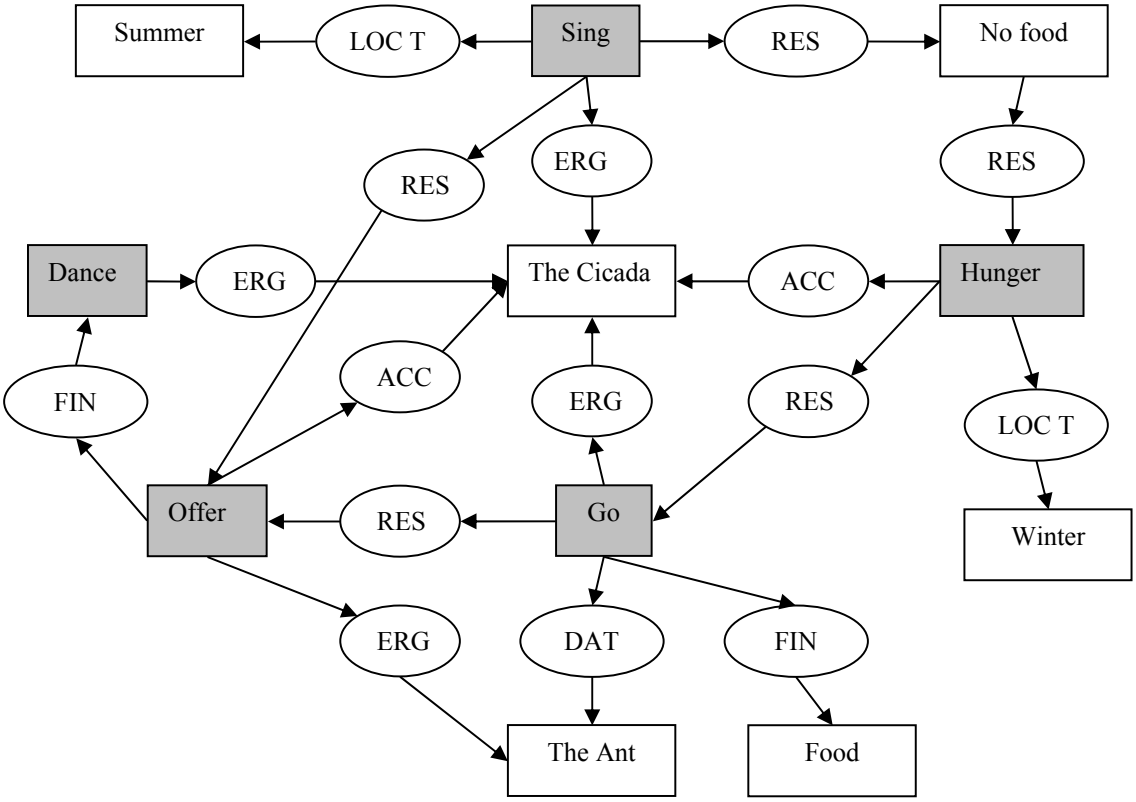
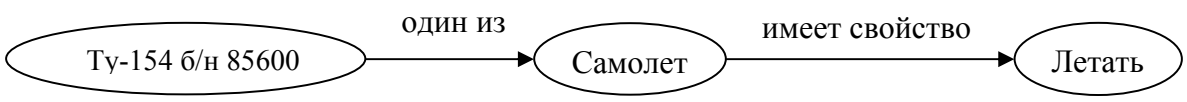


Рис.7.8. Граф Растье для стихотворения "The Cicada and the Ant"

Вследствие голода стрекоза пришла (“Go”) к муравьюс целью (“Fin”) получить еды (“Food”). Муравей предложил стрекозе плясать (“Dance”) вследствие того, что она пришла к нему, и того, что она пела летом.

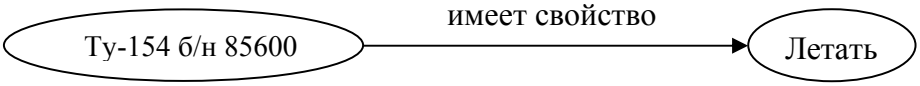
7.7. Проблемы построения семантических сетей

Как показано в подразд.7.1, семантическая сеть должна хранить знания в математически точной форме. В этой связи, ее построение требует аккуратности и хорошего понимания предметной области и всех связанных с ней понятий. Проблемы представления знаний были изложены в уже упомянутой работе Дрю Макдермота [6]. На первый взгляд, построение графов подобных приведенным в подразд. 7.3. примерам может проходить легко и непринужденно. Однако это далеко не всегда так. Рассмотрим следующую конструкцию:



Иными словами, Ту-154 с бортовым номером 85600 является экземпляром

класса самолетов, а самолеты обладают свойством летать. Поскольку отдельный представитель класса наследует признаки класса, мы делаем вывод, что Ту-154 с бортовым номером 85600 тоже может летать,



и будем правы. Но если теперь мы наложим на такую же схему другие факты (очень старый и широко известный парадокс):

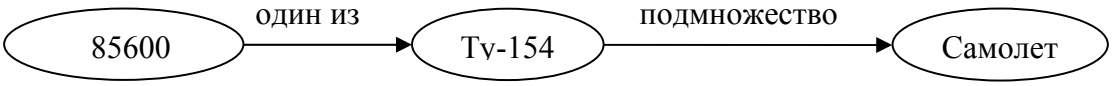
Сократ – один из людей.

Людей много.

Следовательно, Сократов много.

А ведь в отношении самолета это работало! Мы попали в ловушку по следующей причине. Понятия «Самолеты» и «Люди» - это множества самолетов и людей соответственно. Свойство «летать» относится не к множеству, а к его экземплярам. Свойство же «Многочисленность» относится к классу в целом и неприменимо к отдельным экземплярам данного класса. Таким образом, всегда следует четко понимать, что соответствует тому или иному понятию, единичный экземпляр или подмножество, и правильно применять предикаты.

Другая проблема связана с присвоением понятиям имен. В приведенном выше примере с самолетом не зря конкретный самолет был идентифицирован бортовым номером. В противном случае было бы непонятно, какой именно экземпляр самолета имеется в виду.



Для выявления свойства «летать» точная идентификация экземпляра не критична, чего нельзя сказать, если речь идет о необходимости устранении неисправности. Приведенная выше идентификация 85600 также не является исчерпывающей. Во-первых, данная комбинация цифр может относиться к чему угодно, например, номеру телефона. Во-вторых, самолет может попасть в гражданскую авиацию из авиации военной, где идентификация совершенно другая, и мы не сможем ничего узнать о прошлой жизни самолета, например, о предыдущих ремонтах, а это совершенно недопустимо.

Проблема идентификации понятий имеет место и в реальной жизни, но здесь мы мудро пользуемся «бритвой Оккама»: Не умножай сущностей сверх необходимости. Если мы общаемся в небольшой компании, достаточно имен. В студенческой группе можно идентифицировать каждого фамилией. Однофамильцы при этом удостаиваются имени, а тезки – еще и отчества. Если создается семантическая сеть, охватывающая большое количество объектов, то

неизбежно возникает проблема **синонимии**, когда одно имя указывает на различные понятия. И если в древности достаточно было сказать «Иисус из Назарета», чтобы идентифицировать человека, то сейчас даже используемая в паспортном учете триада «ФИО» - «дата рождения» - «место рождения» не гарантирует отсутствия повторяющихся идентификаторов. Кроме того, такой громоздкий ключ (выражаясь в терминах баз данных) не способствует наглядности и простоте восприятия. В этой связи для локальных семантических сетей могут использоваться принятые в практике номера зачетных книжек, табельные номера, ИНН и т.п. Другая проблема – **полисемия**, когда одно слово используется для обозначения различных понятий. Синонимия и полисемия могут катастрофически усложнить проблему построения больших сетей и, в частности, объединение фрагментов, написанных разными авторами.

Название вершины является всего лишь символическим именем, его осмысленность только увеличивает наглядность графа. Полностью идентифицируют вершину ее свойства, например, для человека – фамилия, имя, отчество, дата рождения и т.п.

7.8. Факты и правила в семантической сети

Рассмотренные выше отношения, записанные в виде субъект-предикат-объект, представляют собой неизменные знания, т.е. факты. Занесение всех известных фактов о каждом объекте может потребовать неоправданно много времени. В качестве примера уместно привести родственные отношения. Для любых двух родственников есть название отношения между ними: дядя-племянник, свекровь-зять и т.п. Таким образом, для семьи из n=10 человек число отношений будет равно n*(n-1) = 90. При этом часть отношений являются первичными (супруг-супруга и родитель-ребенок), остальные отношения вытекают из первичных. Если информацию о том, как вторичные отношения определяются на основе первичных, записать в виде правил, то для каждого объекта можно заносить в базу знаний только первичные факты. Для семейных отношений это означает сокращение, не менее, чем, в n/3 раз, если считать, что каждый член семьи является чьим-то ребенком и родителем, а также чьим-то супругом, и не более того.

Одним из стандартов языка представления правил является SWRL - Semantic Web Rule Language (http://www.w3.org/Submission/SWRL/). Данный язык является расширением XML, и конструкции на нем предназначены исключительно для машинной интерпретации. Редакторы правил обычно предоставляют вариант “human readable” правил в формате, подобном правилам на Прологе, для их создания и отладки. Ниже приведен фрагмент правила определения отношения «дядя» для чтения человеком и исходный текст данного правила на языке SWRL.

$$hasParent(?x1,?x2) \wedge hasBrother(?x2,?x3) \Rightarrow hasUncle(?x1,?x3)$$

```
<swrl:Imp rdf:ID="Bros">
  <swrl:body>
```

```
        <swrl:AtomList>
          <rdf:first>
            <rdf:Description>
              <rdf:type rdf:resource="&swrl;ClassAtom"/>
              <swrl:argument1>
                <rdf:Description rdf:about="#x3"/>
              </swrl:argument1>
              <swrl:classPredicate rdf:resource="#Person"/>
            </rdf:Description>
          </rdf:first>
          <rdf:rest>
            <swrl:AtomList>
              <rdf:first>
                <rdf:Description>
                  <rdf:type rdf:resource="&swrl;IndividualPropertyAtom"/>
                  <swrl:argument2>
                    <rdf:Description rdf:about="#x3"/>
…
                  <rdf:Description rdf:about="#x1"/>
                </swrl:argument1>
                <swrl:propertyPredicate rdf:resource="#hasBrother"/>
              </rdf:Description>
            </rdf:first>
            <rdf:rest rdf:resource="&rdf;nil"/>
          </swrl:AtomList>
        </swrl:head>
      </swrl:Imp>
```

Устанавливая правила для объектов семантической сети, мы можем столкнуться с проблемой открытого или закрытого мира (Open or Closed World Assumption). Допущение об открытом мире предполагает, что никто не располагает полной информацией об окружающем мире, следовательно, выводы должны делаться исключительно на основании того, что известно. Допущение о закрытом мире предполагает, что вся информация известна наблюдателю. В качестве примера можно привести родственное отношение типа «мачеха». Пусть в базе знание имеются следующие факты:

Андрей является родителем Егора.

Юлия является супругой Андрея.

В соответствии с допущением закрытого мира, Юлия является мачехой Егора, поскольку в базе нет сведений о том, что она его мать. В открытом мире Юлию можно считать мачехой Егора только в том случае, если известно, что его матерью является не Юлия, а другая женщина.

Применение правил крайне полезно в тех случаях, когда в базе знаний содержится неполная информация. Пусть, например, в предыдущем примере указано, Андрей – это человек, но нет сведений, что Юлия – человек. Тогда все правила наподобие

*Если **X** человек, то **X** имеет фамилию*

не смогут быть применены к Юлии. Если же создать правило

Если ***X*** *человек И* ***X*** *супруг* ***Y*** *ТО* ***Y*** *человек*

то можно будет установить факт, что Юлия тоже человек. Помимо положительного эффекта от использования правил имеется и недостаток: комбинаторная сложность, которая по мере увеличения объема базы знаний довольно быстро вырастает до космических масштабов. Так, например, если в достаточно маленькой базе знаний имеется 100 фактов и 10 правил по три факта в каждом, то общее количество попыток применить правила к фактам может достигать значения $10*100*100*100 = 10^7$, поскольку в каждое правило последовательно будут подставляться все возможные факты. Очевидно, что такая «наивная реализация» поиска в семантической сети нежизнеспособна. Как в любой задаче поиска, здесь нужно решать проблему сокращения комбинаторной сложности. В качестве примера ускорения обработки правил можно привести алгоритм **Rete** (Рити) (http://en.wikipedia.org/wiki/Rete_algorithm), основной смысл которого заключается в том, что строится дерево, каждый узел которого соответствует части условий правил и хранит список фактов, удовлетворяющих этим условиям. Поскольку в ходе применения правил постоянно возникают новые факты, они прогоняются по сети, и списки фактов при вершинах обновляются. Узким местом алгоритма **Rete** является большой требуемый объем памяти, поскольку одни и те же факты многократно дублируются в списках при вершинах графа.

В качестве одного из альтернативных путей решения можно запускать все возможные правила для каждого документа один раз и сохранять результаты в виде фактов. Для базы родственных связей это будет означать, что в документ вначале заносятся только первичные связи (родители-дети и супруги), затем из них вычисляются все опосредованные отношения (внучатные племянники и т.д.), которые на равных правах затем пополняют базу знаний. После этого все факты будут извлекаться одинаково быстро. Такой подход является разновидностью вывода на основе прецедентов (Case based reasoning), и его можно считать аналогом навыков в человеческом интеллекте. На самом деле, мы почти всегда действуем по аналогиям, например, в устной речи. Если бы мы при построении каждой фразы применяли правила языка, то скорость речи не превышала бы нескольких предложений в час. Это особенно заметно, когда мы переводим русский текст на иностранный язык. Если языковая конструкция нам знакома (многократно использовалась ранее), то перевод идет в быстром темпе. Если мы создаем предложение в первый раз, то процесс перевода замедляется в десятки и сотни раз.

7.9. Интеллектуальный агент семантической сети

Построить семантическую сеть – задача непростая. Но после того, как мы с ней справимся, возникнет вопрос, а как извлекать знания из семантической сети? Очевидно, для этой цели должна быть разработана специальная программа, которая на основе запроса пользователя проведет поиск требуемых

знаний и выдаст результат.

В настоящее время существует несколько языков запросов к базам знаний в виде семантических сетей в формате RDF, в частности, DQL, R-DEVICE, RDFQ, RDQ, RDQL, SeRQL. Наиболее стандартизованным является язык SPARQL, прошедший стандартизацию в группе Data Access Working Group (DAWG) консорциума World Wide Web (W3C). Существуют несколько реализаций языка SPARQL для различных программных платформ. Автор протестировал некоторые из них, и оказалось, что запросы на языке SPARQL обрабатывают только факты (триплеты субъект-предикат-объект), но не понимают правил. Тем самым вся работа по созданию онтологий становится бессмысленной.

Для устранения этого недостатка автор разработал упрощенный язык представления семантических документов и программу, поддерживающую визуализацию знаний и выполнение простейших запросов. Программа SEMANTIC, предлагаемая в рамках данной дисциплины в качестве оболочки для создания и исследования семантических сетей, содержит зачатки свойств такого интеллектуального агента. В частности, программа применяет ко всем фактам, записанным в базу знаний, правила наследования, а также позволяет пользователю создавать собственные правила. Более подробная информация об этой программе содержится в Приложении 3.

7.10. Управление контекстом

Необходимость однозначно идентифицировать все объекты семантической сети приводит не только к усложнению процедуры добавления фактов, но и к тому, что извлечение знаний становится очень громоздким. Упростить понимание этой проблемы можно не простом примере. Пусть мы хотим на денек попросить у соседа конспект лекций по искусственному интеллекту, который он, в свою очередь, одолжил у своей подружки. Тогда диалог будет приблизительно следующим:

«Гражданин Российской Федерации Сидоров Владимир Иванович, родившийся в 1985 году в г. Саратове, имеющий паспорт № 60 04 123456, выданный 20.05.2003 51-м ОМ г. Санкт-Петербурга, дай мне, гражданину Российской Федерации Петрову Ивану Викторовичу, родившемуся в 22.04.1986г. в г. Пскове, имеющему паспорт № 6606 654321, на 24 часа 00 минут 00 секунд конспект лекций по дисциплине «Искусственный интеллект», который читает к.т.н., доцент кафедры вычислительной техники Бессмертный Игорь Александрович, ...».

Фраза, немыслимая в повседневной ситуации, но совершенно нормальная в милицейском протоколе.

Очевидно, что при создании семантической сети один раз можно постараться и идентифицировать все объекты однозначно, хотя это существенно усложнит работу. Но для доступа к знаниям необходимо дать возможность вести упрощенный диалог, подобный имеющему место в реальной

жизни. Такая функция может быть возложена на интеллектуальный агент, осуществляющий доступ к знаниям.

База контекста должна состоять из двух компонентов: постоянного и временного. Постоянный контекст – это знания, не изменяющиеся в процессе диалога. Например, мы хотим узнать, который час. На этот вопрос, который ни у кого не вызывает затруднений, не может быть получен ответ без информации о местоположении субъекта. Следовательно, в базе контекста должна быть информация о том, где находится субъект, а также часовой пояс данного места. Иными словами, в базу должно быть загружено контекстное окружение.

Временный контекст – это факты, которые устанавливаются или уничтожаются (забываются) в процессе диалога, а также временные ассоциации, устанавливаемые для упрощения диалога. Временные факты – это, например, ответы на вопросы, которые были заданы ранее, т.е. знания, принесенные извне и не требующие сохранения в базе знаний. Примером таких фактов могут быть ответы пациента на вопросы врача, который пытается поставить диагноз. Отсутствие такой памяти сделает диалог похожим на многочисленные анекдоты про склеротиков. Временные ассоциации дают возможность присвоить объектам или фактам короткие имена для использования только в данном диалоге. Временные ассоциации широко используются как в повседневной жизни, так и в документах. Например, в текстах договоров обычно используется оборот типа «ООО РОГА И КОПЫТА в лице директора Фунта А.А., действующего на основании устава, именуемое в дальнейшем ПОКУПАТЕЛЬ…».

Таким образом, база контекста позволит создать для пользователя упрощенное представление (модель) семантической сети, которое позволит вести диалог в привычном виде.

7.11. Семантическая сеть и Семантическая паутина

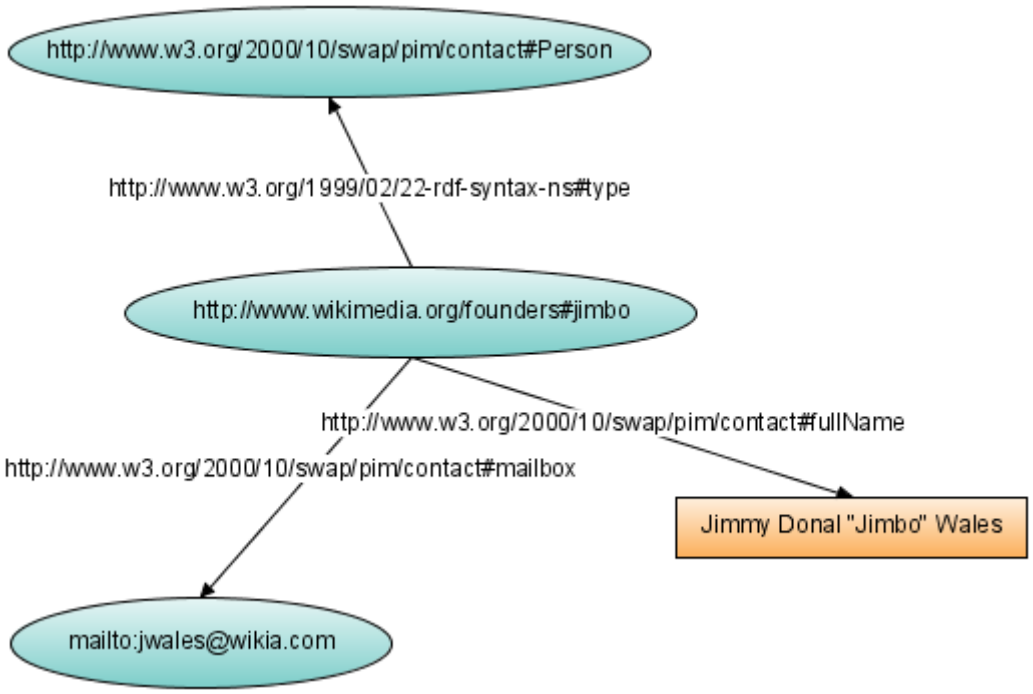
Как показано в подразд.7.2, семантическая сеть имеет давнюю историю, и до последнего времени это понятие не вызывало никаких двусмысленностей. Однако, после того, как изобретатель Всемирной паутины (WWW) Тим Бернерс Ли в 2001 году провозгласил концепцию “Semantic Web”, в русскоязычной литературе эти два понятия стали смешиваться. Для устранения недоразумений предлагается переводить понятие “Semantic Web” как «Семантическая Паутина» или, в данном тексте – просто «Паутина». Суть идеи Тима Бернерса Ли заключается в том, чтобы снабдить ресурсы Интернета специальными метаданными, доступными для компьютерной обработки и однозначно характеризующими свойства и содержание ресурсов Всемирной паутины, вместо используемого в настоящее время текстового анализа документов. В частности, предполагается снабдить все ресурсы Интернета тегами, позволяющими установить автора каждого документа. Следует отметить, что это решение сделано «вдогонку» к концепции WWW после того, как Интернет

из хранилища знаний превратился в информационную «помойку» из-за допускаемой анонимности ресурсов. Однако, данное свойство семантической паутины нас интересует в последнюю очередь. В центре нашего интереса – свойства Семантической Паутины доставлять пользователю информацию, которую он хочет получить.

Основное отличие Семантической Паутины от WWW заключается в том, что связи между ресурсами WWW отражают размещение документов, а связи Паутины – содержание. Правда, как в WWW, так и в Паутине ссылки содержат адреса ресурсов, но в WWW ссылки обезличенные; их смысл должен понимать пользователь. В Паутине же ссылки имеют явно указанный смысл, который доступен для машинной обработки. Таким образом, Семантическая Паутина подразумевает роботизированный поиск информации вместо имеющего место сейчас Web сёрфинга, когда пользователь Интернета переходит по ссылкам от документа к документу.

7.12. Семантическая Паутина: принципы и текущее состояние

В основе Семантической Паутины лежит тот же триплет субъект-предикат-объект. Отличие заключается в представлении каждого из элементов триплета. В Паутине субъект, объект и предикат представлены универсальными идентификаторами ресурсов или URI (Uniform Resource Identifier). Так, граф визитной карточки основателя Википедии выглядит следующим образом (рисунок позаимствован в той же Википедии):



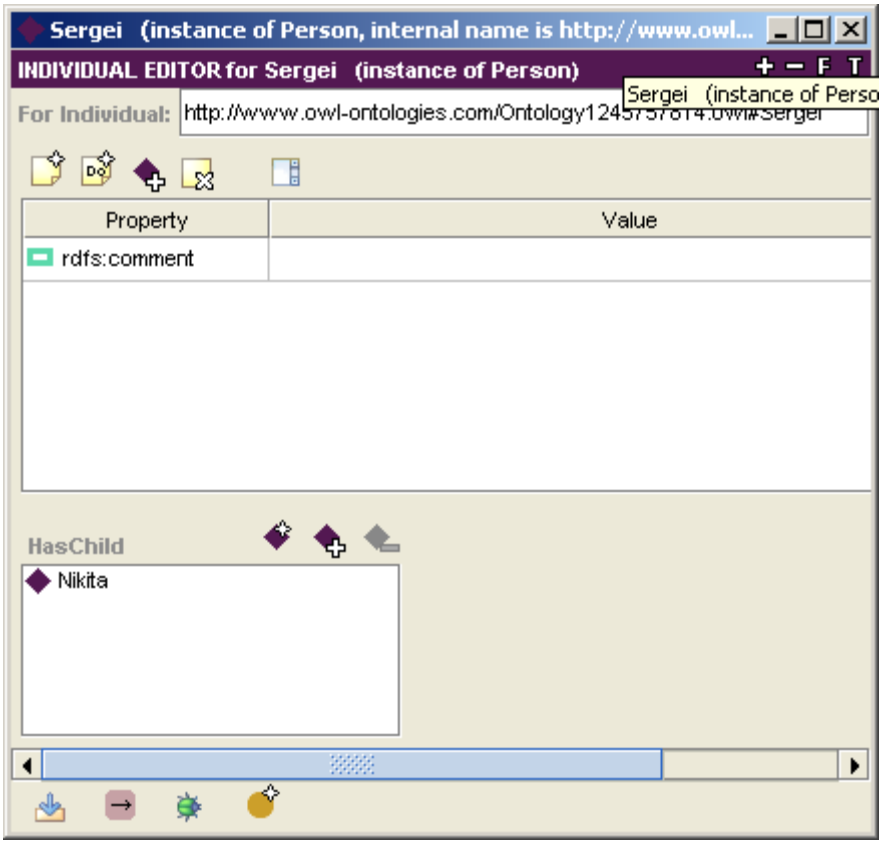
URI только выглядит как адрес ресурса URL. На самом деле по любому из приведенных на рисунке адресов, таких как

http://www.w3.org/2000/10/swap/pim/contact#mailbox, вы не найдете ничего похожего на почтовый ящик, или хотя бы его описание, как это может показаться на первый взгляд. Такая система именования нужна только для обеспечения уникальности имен.

Наиболее популярным языком представления знаний в Паутине является язык RDF. Фрагмент RDF-документа, описывающие отношения Сергей имеет ребенка Никиту, а Никита имеет родителя Сергея, приведен ниже.

```
<Person rdf:ID="Nikita">
   <HasParent rdf:resource="#Sergei"/>
</Person>
<owl:Class rdf:ID="Person"/>
<Person rdf:ID="Sergei">
   <HasChild rdf:resource="#Nikita"/>
</Person>
```

Данный формат не предназначен для чтения человеком. Редакторы семантических документов, например, наиболее распространенный редактор онтологий Protégé (http://protege.stanford.edu/), дают возможность вводить и редактировать факты в экранных формах (фреймах), как показано ниже.



В настоящее время создание RDF/OWL документов ведется отдельными энтузиастами. Количество документов в Интернете невелико, и их можно найти

с помощью специального поискового сервера, например, SWOOGLE (http://swoogle.umbc.edu/). Развитие проекта сдерживается по многим причинам. В основном это отсутствие быстрой и прямой выгоды от создания семантических ресурсов, сложность формализации знаний и отсутствие универсальных агентов для извлечения знаний. Представленные в Интернете реализации интеллектуальных агентов ограничиваются студенческими разработками вроде путеводителя по пивным города Саутхемптон (http://www.twine.com/item/11by40gxk-p1/southampton-pub-guide-in-rdf) и винного агента, советующего, какое вино лучше употребить с каждым из блюд (http://ksl.stanford.edu/people/dlm/webont/wineAgent/).

8. Домашние задания и лабораторные работы

8.1. Домашнее задание №1. Изучение работы Prolog программы

Цель домашнего задания: изучение работы Prolog программы на примере решения задачи сортировки двумя методами: сортировка методом вставки и так называемая «быстрая» сортировка. Текст обоих программ приведен ниже.

```
/************** Сортировка методом вставки   *************/
    insrtsort([],[]).
    insrtsort([Head|Tail], ListSorted) :- insrtsort(Tail,TailSorted),
      insrt(Head,TailSorted,ListSorted).
    insrt(X, [Y | ListSorted], [Y | ListSorted1]) :-
        X > Y, !, insrt(X, ListSorted, ListSorted1).
    insrt(X,ListSorted, [X | ListSorted]).
/************** Быстрая сортировка    *****************/
    fastsort([],[]).
    fastsort([Head | Tail], ListSorted) :-
      split(Head, Tail, TailLess, TailGreater),
      fastsort(TailLess,TailLessSorted),
      fastsort(TailGreater,TailGreaterSorted),
      append(TailLessSorted, [Head | TailGreaterSorted], ListSorted).
    split(_, [], [], []).
    split(X, [H | T], [H | TL], TG) :- H < X, !, split(X, T, TL, TG).
    split(X, [H | T], TL, [H | TG]) :- split(X, T, TL, TG).
```

В качестве исходных данных использовать собственную дату рождения в следующем виде: [D,D,M,M,Y,Y,Y,Y], например, дата рождения 20.09.1984. Сортируемый список будет [2,0,0,9,1,9,8,4].

Требуется:

1) запустить каждую из программ на выполнение в режиме трассировки;
2) записать структурированную трассу программы, отражающую рекурсивную вложенность предикатов;
3) описать логику предикатов в каждой из программ;
4) выбрать критерии и сравнить два метода сортировки.

Содержание отчета:

1) исходный текст программы с комментариями;
2) структурированная трасса хода выполнения каждой из двух программы;
3) описание логики каждого из методов сортировки;
4) сравнение рассмотренных методов сортировки.

8.2. Домашнее задание №2. Изучение алгоритмов поиска

Цель задания: Исследование алгоритмов решения задач методом поиска.

Описание предметной области. Имеется транспортная сеть, связывающая города СНГ. Сеть представлена в виде таблицы связей между городами. Связи являются двусторонними, т.е. допускают движение в обоих направлениях. Необходимо проложить маршрут из одной заданной точки в другую.

Этап 1. Неинформированный поиск. На этом этапе известна только топология связей между городами. Выполнить:

1) поиск в ширину;
2) поиск глубину;
3) поиск с ограничением глубины;
4) поиск с итеративным углублением;
5) двунаправленный поиск.

Отобразить движение по дереву на его графе с указанием сложности каждого вида поиска. Сделать выводы.

Этап 2. Информированный поиск. Воспользовавшись информацией о протяженности связей от текущего узла, выполнить:

1) жадный поиск по первому наилучшему соответствию;
2) затем, использую информацию о расстоянии до цели по прямой от каждого узла, выполнить поиск методом минимизации суммарной оценки A*.

Отобразить на графе выбранный маршрут и сравнить его сложность с неинформированным поиском. Сделать выводы.

Таблица связей между городами

Город 1	Город 2	Расстояние, км
Вильнюс	Брест	531
Витебск	Брест	638
Витебск	Вильюс	360
Воронеж	Витебск	869
Воронеж	Волгоград	581
Волгоград	Витебск	1455
Витебск	Ниж.Новгород	911
Вильнюс	Даугавпилс	211
Калининград	Брест	699
Калиниград	Вильнюс	333
Каунас	Вильнюс	102
Киев	Вильнюс	734
Киев	Житомир	131
Житомир	Донецк	863

Житомир	Волгоград	1493
Кишинев	Киев	467
Кишинев	Донецк	812
С.Петербург	Витебск	602
С.Петербург	Калининград	739
С.Петербург	Рига	641
Москва	Казань	815
Москва	Ниж.Новгород	411
Москва	Минск	690
Москва	Донецк	1084
Москва	С.Петербург	664
Мурманск	С.Петербург	1412
Мурманск	Минск	2238
Орел	Витебск	522
Орел	Донецк	709
Орел	Москва	368
Одесса	Киев	487
Рига	Каунас	267
Таллинн	Рига	308
Харьков	Киев	471
Харьков	Симферополь	639
Ярославль	Воронеж	739
Ярославль	Минск	940
Уфа	Казань	525
Уфа	Самара	461

Варианты заданий:

Номер варианта	Исходный пункт	Пункт назначения
1	Мурманск	Одесса
2	С.Петербург	Житомир
3	Самара	Ярославль
4	Рига	Уфа
5	Казань	Таллин
6	Симферополь	Мурманск
7	Рига	Одесса
8	Вильнюс	Одесса
9	Брест	Казань
10	Харьков	Ниж.Новгород

Расстояние до цели по прямой взять из подходящей географической карты.

Правило выбора варианта задания. Дату рождения подвергнуть следующей свертке: (ДД + ММ) mod 10 + 1 = номер варианта.

8.3. Домашнее задание №4. Расчет сети Байеса

Цель: Расчет полных и условных вероятностей для различных событий.

Предметная область: характеристики студентов Вашего потока.

Исходные данные. Выбрать 4 атрибута, характеризующих студентов, и заполнить таблицу следующего вида:

Ф.И.О.	Получает стипендию	Живет в общежитии	Подрабатывает	Занимается спортом
Иванов	Нет	Да	Нет	Нет
…	…	…	…	…

Вместо приведенных в качестве примера атрибутов (получает стипендию, живет в общежитии,…) подставить атрибуты, выбранные самостоятельно.

Объем таблицы – не менее 25 строк. Заполнение таблицы строго индивидуально. У каждого должен быть свой вариант заполнения.

Провести расчеты с использованием сети Байеса.

1. Выбрать зависимые и независимые переменные. Составить сеть Байеса для данного набора переменных. Вычислить все априорные, условные и полные вероятности.
2. Вычислить информативность каждого атрибута по формуле Шеннона. Построить дерево решений, как в разд. 5.3, руководствуюсь данными об информативности атрибутов. Сравнить полученное дерево с выбранным в п.1. разбиением на зависимые и независимые переменные.
3. Добавить в таблицу 5 строк. Пересчитать. Сделать выводы о достаточности (недостаточности) данных.

Содержание отчета:

1. Исходная таблица.
2. Граф сети Байеса со всеми вероятностями.
3. Показатели информативности по Шеннону
4. Дерево решений с вероятностями.
5. Выводы.

8.4. Лабораторная работа №1. Прогнозирование с помощью нейронной сети

Цель работы: Исследование программы NeuroGenetic Optimizer (NGO).

Задание: Загрузить в программу NGO котировки акций в соответствии с вариантом:

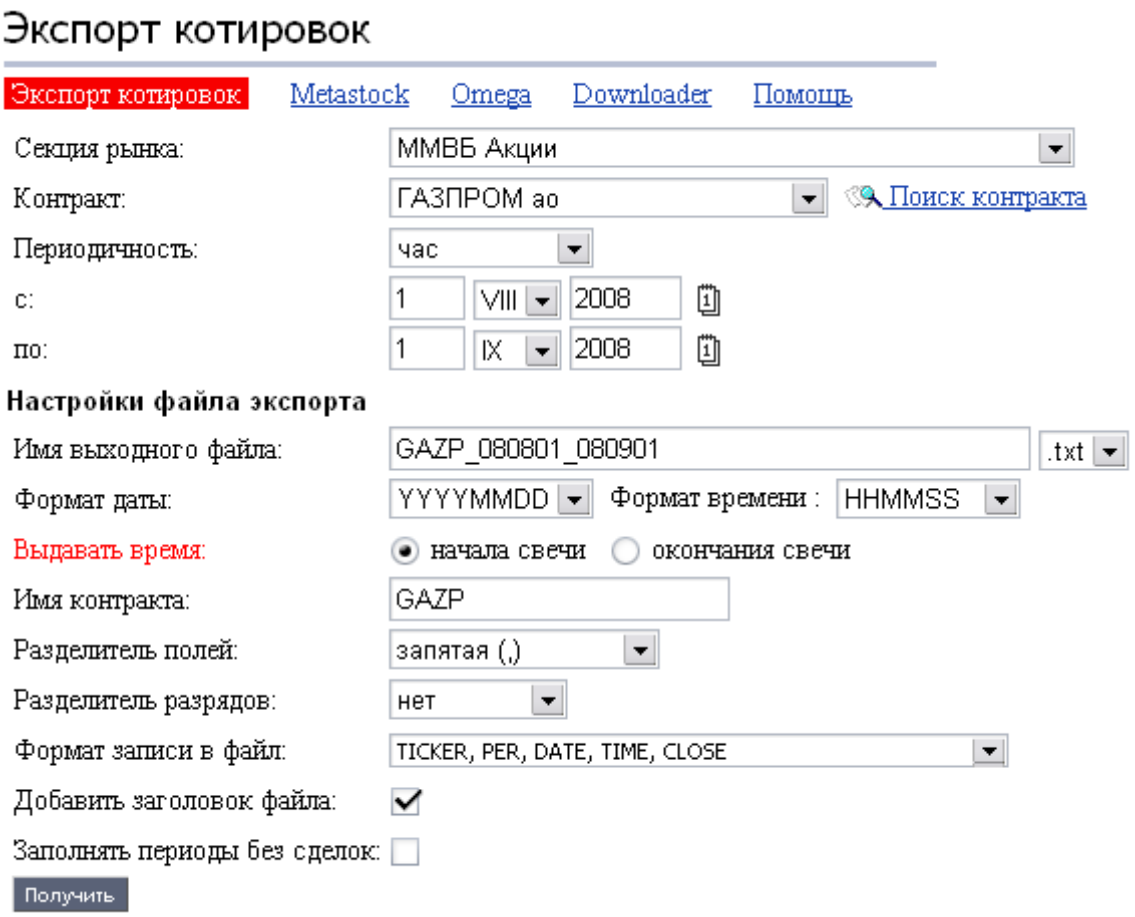
Номер варианта	Код	Наименование
1	GAZP	АО Газпром
2	ГМКНорНик	Норильский никель
3	ЛУКОЙЛ	НК Лукойл
4	МТС-ао	Компания МТС – обыкн. акции
5	Роснефть	Роснефть
6	Ростел-ао	Ростелеком – обыкн.акции
7	Сбербанк	Сбербанк – обыкн.акции
8	Сургнфгс	Сургутнефтегаз
9	УРалСвИ-ао	Уралсвязинформ – обыкн.акции
10	Уралкалий-ао	АО Уралкалий – обыкн.акции

Номер варианта выбирается так же, как в домашнем задании №2.

Для получения исходных данных войти на сайт

http://www.finam.ru/analysis/export/default.asp

Выбрать контракт (эмитента акций) в соответствии с вариантом, период не менее полугода, имя файла – на свое усмотрение. Остальные параметры задать, как указано на следующем снимке экрана.



Экспортированный файл будет иметь следующий вид (на примере Газпрома).

<TICKER>,<PER>,<DATE>,<TIME>,<CLOSE>
GAZP,60,20080801,100000,277.80000
GAZP,60,20080801,110000,278.20000
GAZP,60,20080801,120000,275.89000
GAZP,60,20080801,130000,274.44000
GAZP,60,20080801,140000,273.90000
GAZP,60,20080801,150000,272.05000
GAZP,60,20080801,160000,275.50000
GAZP,60,20080801,170000,273.22000
GAZP,60,20080804,100000,269.70000
GAZP,60,20080804,110000,269.38000
GAZP,60,20080804,120000,269.74000
GAZP,60,20080804,130000,267.69000
GAZP,60,20080804,140000,267.78000
GAZP,60,20080804,150000,267.19000
GAZP,60,20080804,160000,265.46000
GAZP,60,20080804,170000,263.40000
GAZP,60,20080805,100000,259.46000
GAZP,60,20080805,110000,257.70000
GAZP,60,20080805,120000,252.36000

Экспортированный файл за полгода должен иметь около 1000 строк (8 часов * 5 дней * 25 недель). Для целей прогнозирования использовать **только последнее поле** (цена закрытия).

Выполнить обучение нейронной сети для прогнозирования котировок на следующий час. Использовать временной лаг для порождения массива котировок таким образом, чтобы входными переменными для нейронной сети были значения котировок C_i, C_{i-1}, C_{i-2}, …, а прогнозироваться должно значение C_{i+1}. В приведенной ниже таблице показан фрагмент массива данных для обучения, полученный из файла котировок акций Газпрома.

Входные переменные					Выход
C_{i-4}	C_{i-3}	C_{i-2}	C_{i-1}	C_i	C_{i+1}
277.80	278.20	275.89	274.44	273.90	272.05
278.20	275.89	274.44	273.90	272.05	275.50
275.89	274.44	273.90	272.05	275.50	273.22
274.44	273.90	272.05	275.50	273.22	269.70

Таким образом, прогноз на следующий час строится на основе последовательности котировок за предыдущие пять часов.

Содержание отчета:
1. Описание проблемы.
2. График изменения котировок.
3. Выбор способа деления массива данных на наборы для обучения и для тестирования.
3. Скриншоты работы программы NGO в процессе обучения.
4. Описание построенной нейронной сети (число нейронов, количество слоев, типы функций активации).
5.Анализ точности построения и валидация (оценка практической применимости) полученной нейросетевой модели.

8.5. Лабораторная работа № 2. «Создание информационной системы на базе семантической сети»

Цель работы: изучение семантической сети как инструмента создания информационных и обучающих систем? А также исследование методов логического вывода на основе правил.
Содержание работы:

Выбрать предметную область.

1. Выбрать способ представления знаний в семантической сети – реляционный граф или граф с центром в глаголе, а также язык представления знаний, русский или иной. Возможно многоязычное представление знаний.
2. Записать факты, составляющие предметную область в нотации программы “Semantic”. Рекомендуемый объем базы знаний – не менее 50 фактов.
3. Снабдить базу знаний онтологиями, в т.ч. правилами (не менее 20), позволяющими извлекать новые факты, а также словарями для поддержки диалога на упрощенном естественном языке.
4. Провести тестирование базы знаний, т.е. убедиться в том, что все правила корректно создают новые факты.

Содержание отчета:

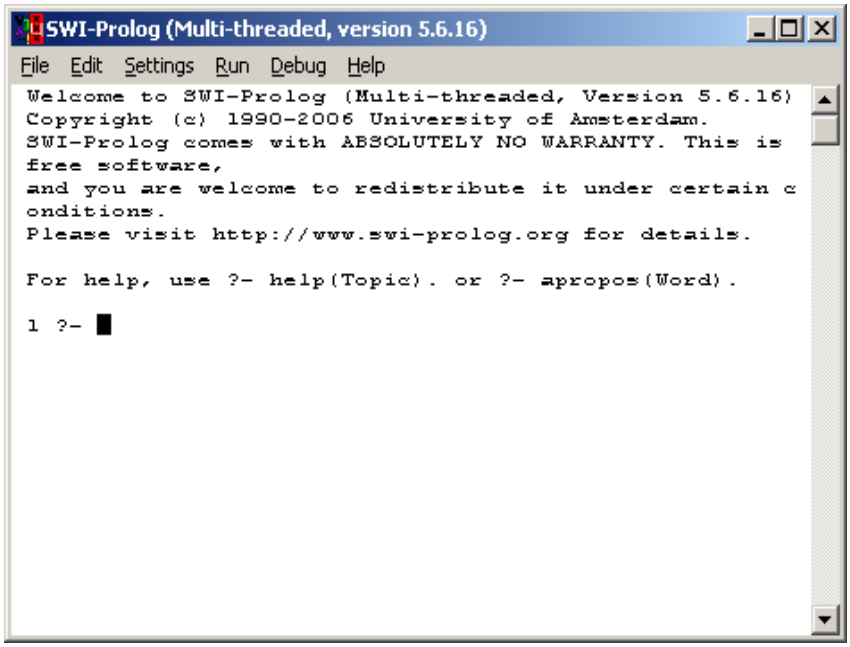
Цель и назначение разработанной информационной системы. Описание предметной области и словарей. Распечатки файлов базы знаний. Снимки экранов одного из вариантов развития диалога.

Литература

1. Стюарт Рассел, Питер Норвиг. Искусственный интеллект:Современный подход. 2-е изд.: пер. с англ. – М.: Изд.дом «Вильямс», 2006. – 1408с.: ил. Парал.тит.англ.
2. www.swi-prolog.org**.** Официальный сайт разработчиков транслятора SWI-Prolog.
3. www.pdc.dk. Официальный сайт компании Prolog Development Center.
4. http://ru.wikipedia.org/wiki/Семантическая_сеть
5. http://compzed.narod.ru/semseti.htm
6. McDermott, Drew. “Artificial Intelligence Meets Natural Stupidity”, **SIGART Newsletter**, No.57 (April, 1976), pp. 4-9.
7. T. Berners-Lee, J. Hendler, and O. Lassila, “The Semantic Web,” **Scientific American**, May, 2001, pp. 34–43.

Приложение 1. Описание программы SWI-Prolog

Исполняемый файл программы SWI-Prolog размещается в **..\bin\plwin.exe**. После запуска появляется окно следующего вида:



Для загрузки программы следует вызвать пункт меню **“File / Consult…”** и выбрать файл с текстом программы, который может быть создан в текстовом редакторе Notepad. Расширение файла по умолчанию .PL. Файл можно создавать и в оболочке программы SWI-Prolog пунктом меню **“File / New…”.** По умолчанию вызывается редактор Notepad (Блокнот). В составе пакета SWI-Prolog имеется более продвинутый редактор (PCE_EMAX), анализирующий синтаксис Пролог-программы, позволяющий устанавливать точки прерываний и т.п. Чтобы включить редактор PCE_EMAX, необходимо в файле **pl.ini** удалить комментарии в строке

:- set_prolog_flag(editor, pce_emacs).

Сделать это можно любым текстовым редактором, либо вызвав пункт меню **“Settings / User init file…”**.

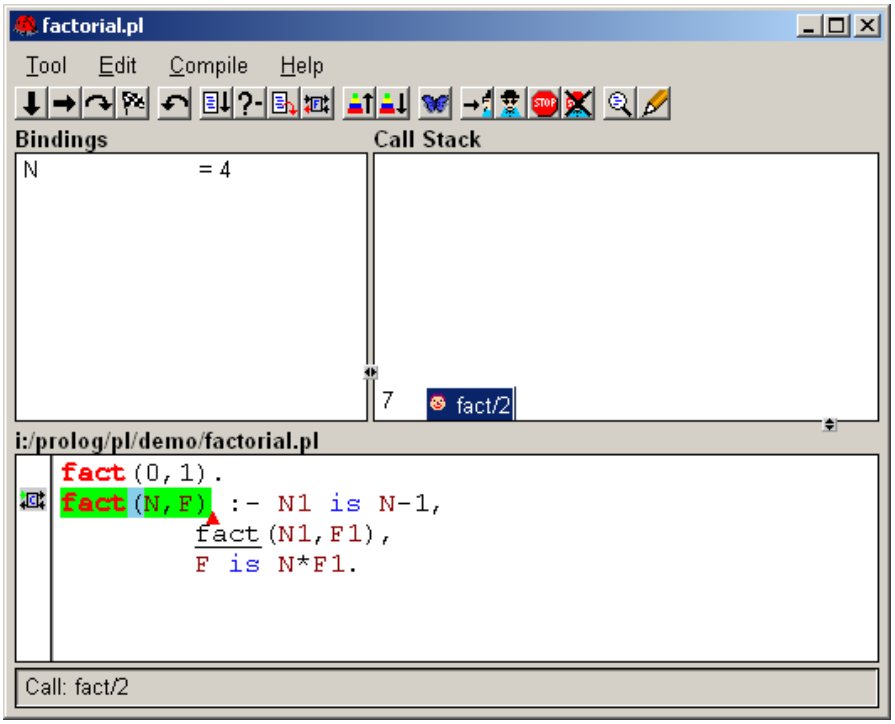
После загрузки в командной строке можно вводить цели. Рассмотрим работу оболочки SWI-Prolog на примере программы вычисления факториала:

```
fact(0,1).
fact(N,F) :- N1 is N-1,
        fact(N1,F1),
        F is N*F1.
```

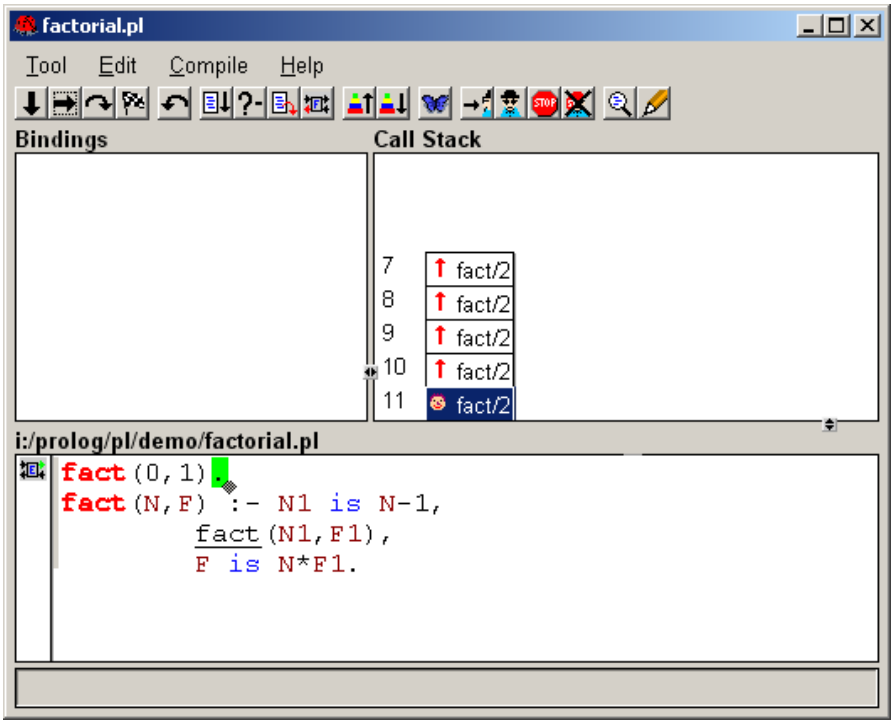
Введем в основном окне программы цель: fact(4,F). Программа выдаст ответ: A = 24. В случае ошибок на этапе трансляции либо выполнения Пролог в этом же окне выдает диагностические сообщения. Пользоваться таким

режимом неудобно. Лучше воспользоваться отладчиком. Для этого необходимо включить опцию графического отладчика **“Debug / Graphical debugger”,** затем с помощью пункта меню **“Debug / Edit spy points…”** включить точку перехвата на интересующий нас предикат, например, в нашем случае на предикат fact, и нажать кнопку с изображением шпиона.

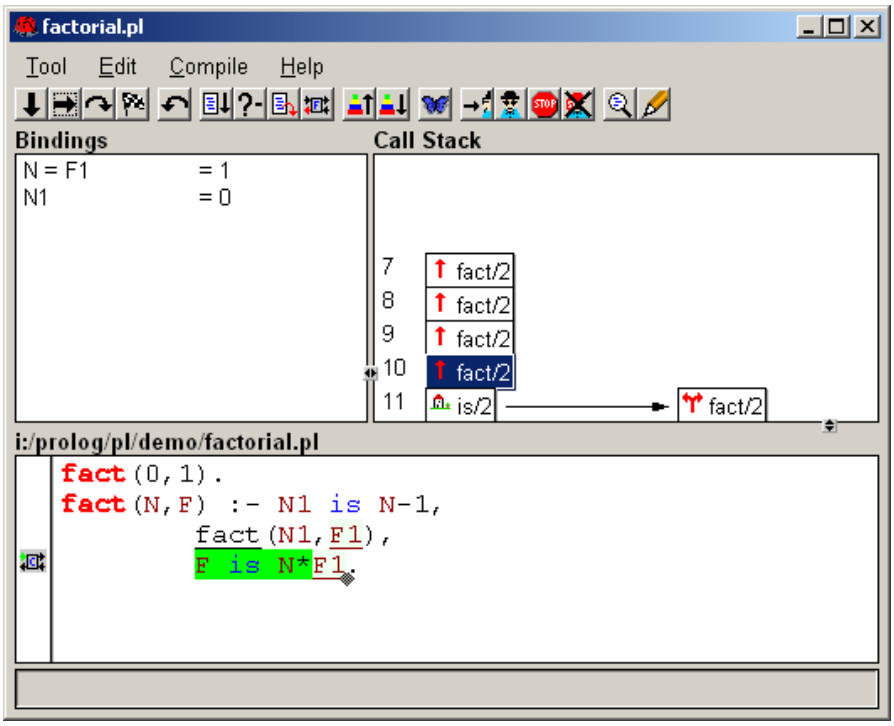
После этого нужно в окне Пролога задать цель, и появится окно отладки, которое выглядит следующим образом:



Панель **Bindings** отображает текущие значения переменных, **Call Stack** – состояние стека, т.е. глубину вложенности рекурсий, наконец, нижняя панель – текст программы, где зеленым цветом выделяется выполняемый предикат. Для пошагового выполнения программы нужно нажимать на кнопку со стрелкой вправо. После максимального углублению в рекурсию стек выглядит следующим образом:



После этого Пролог начинает выполнять собирать факториал из запомненных в стеке значений 1!, 2! и 3!.



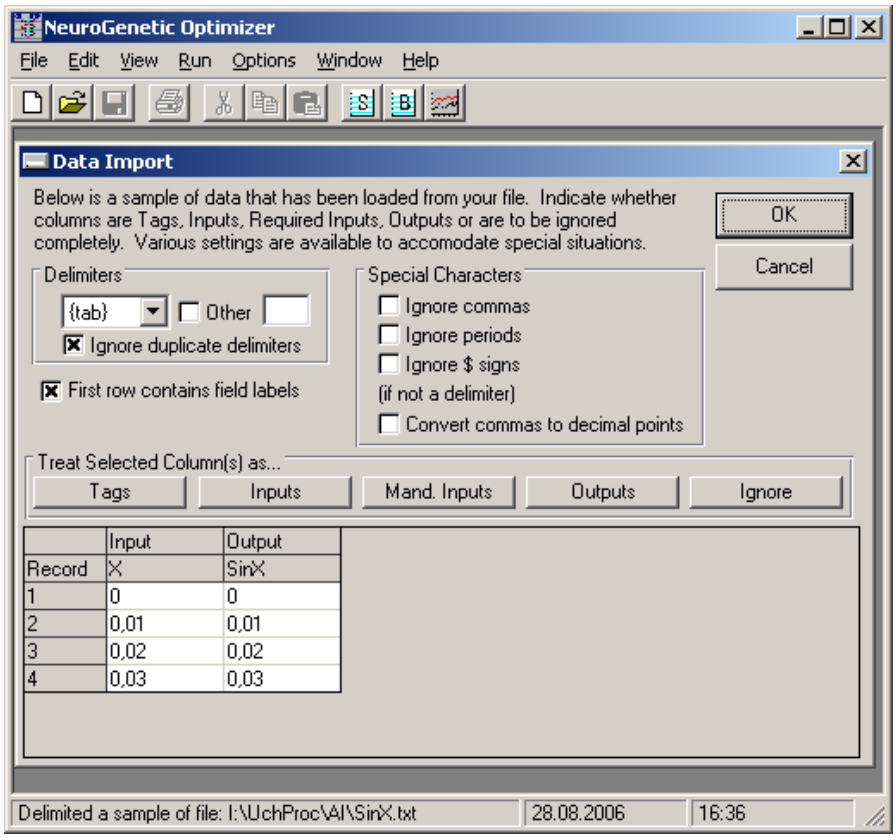
Таким образом, мы можем отслеживать логику работы предикатов и выявлять ошибки.

Приложение 2. Описание программы NGO (NeuroGenetic Optimizer)

Программа NGO предназначена для нейросетевого моделирования на основе обучения. На вход может подаваться до 500 переменных (в зависимости от лицензии). Бесплатный вариант программы поддерживает до пяти входных переменных. Программа состоит из двух компонентов: программы обучения NGO32260.EXE и программы прогнозирования PRE32150.EXE. Для начала работы с программой обучения необходимо загрузить данные. Рассмотрим порядок подготовки работы NGO на примере ее обучения на вычисление функции синуса. Для этого мы должны подготовить файл, содержащий данные в таком виде (разделители – табуляция):

X	Sin(X)
0,01	0,01
0,02	0,019999
0,03	0,029996
0,04	0,039989

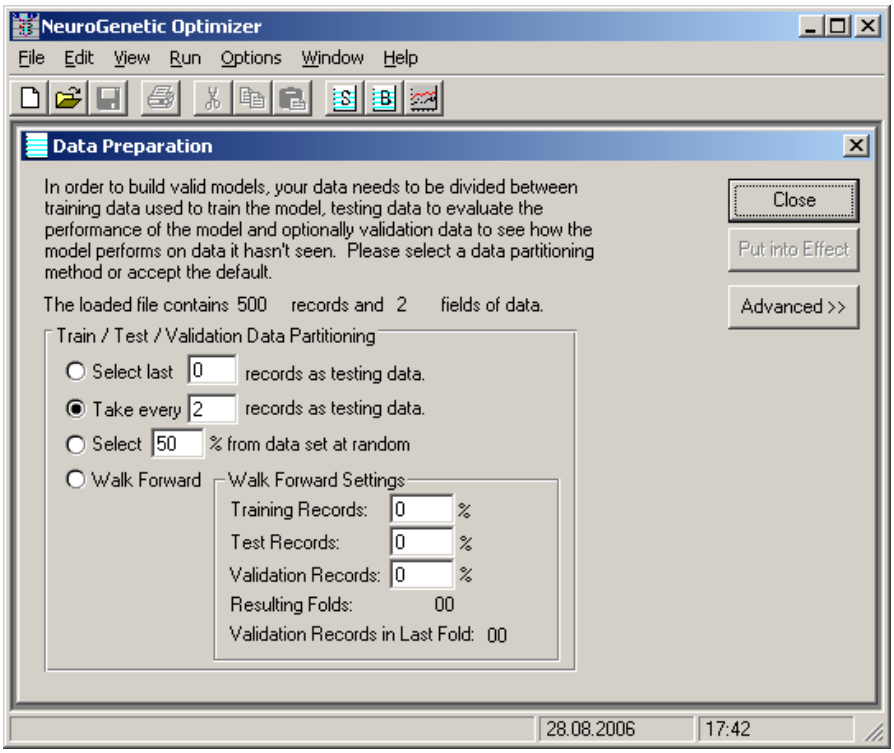
Пусть такой файл содержит 500 строк и охватывает значения X от 0 до 5. Чтобы его загрузить, необходимо выбрать пункты меню File / Open / Data / Train/Test… и местоположение текстового файла. Появится окно следующего вида:



Необходимо указать, что разделителями являются символы табуляции (Delimiters), а также указать назначение переменных (Input, Output). Если переменные не используются, например, их необходимо игнорировать (Ignore). Если в таблице в нижней части окна первые строки данных пустые, значит, NGO не понимает данный формат. Нужно проверить исходный файл.

Важное замечание. Входные данные не подвергаются полному контролю. В частности, NGO не допускает наличия пустых строк в исходном файле, но не диагностирует эту ситуацию. В этом случае программа аварийно завершается на этапе обучения.

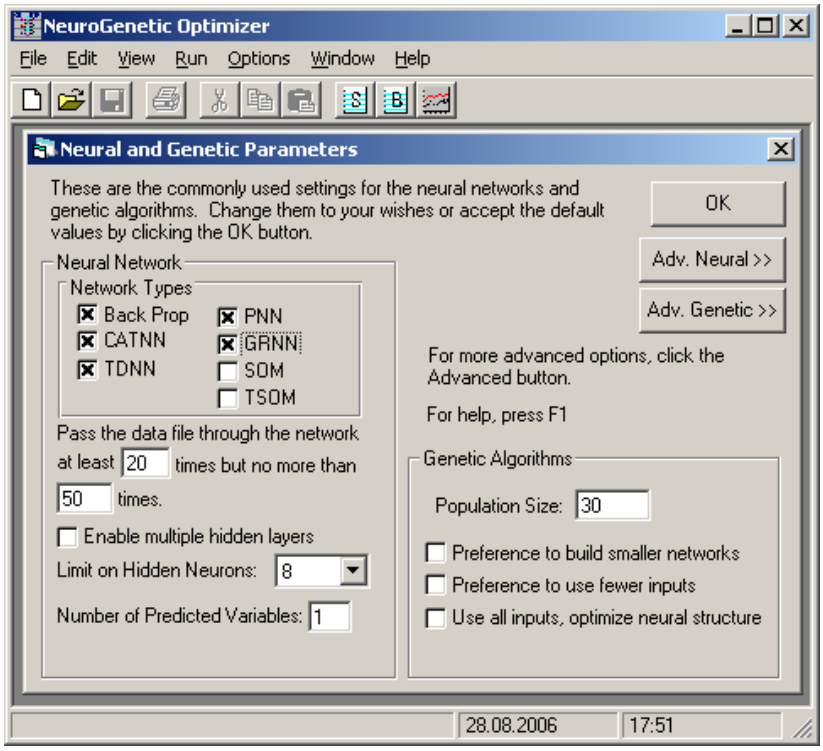
После ввода исходных данных необходимо настроить программу. Для этого нужно вызвать пункт меню Option / Data…



Здесь следует определить, как исходные данные будут делиться на набор для обучения и набор для тестирования. На снимке экрана показано, что для тестирования будет использоваться каждая вторая строка данных. Тестовый набор данных позволяет убедиться в том, что обученная нейронная сеть функционирует правильно, поскольку данные тестового набора в обучении не использовались. В условиях недостаточного количества наблюдений выбор объема тестового набора является критичным: слишком много данных для тестирования (как в данном случае 50 на 50) слишком уменьшает набор для обучения. Если мы сделаем упор на обучение, например, 10 к 1, то результаты тестирования будут ненадежными. Можно для тестирования брать не прореженный набор, а определенное число последних записей. Все зависит от

характера исходных данных.

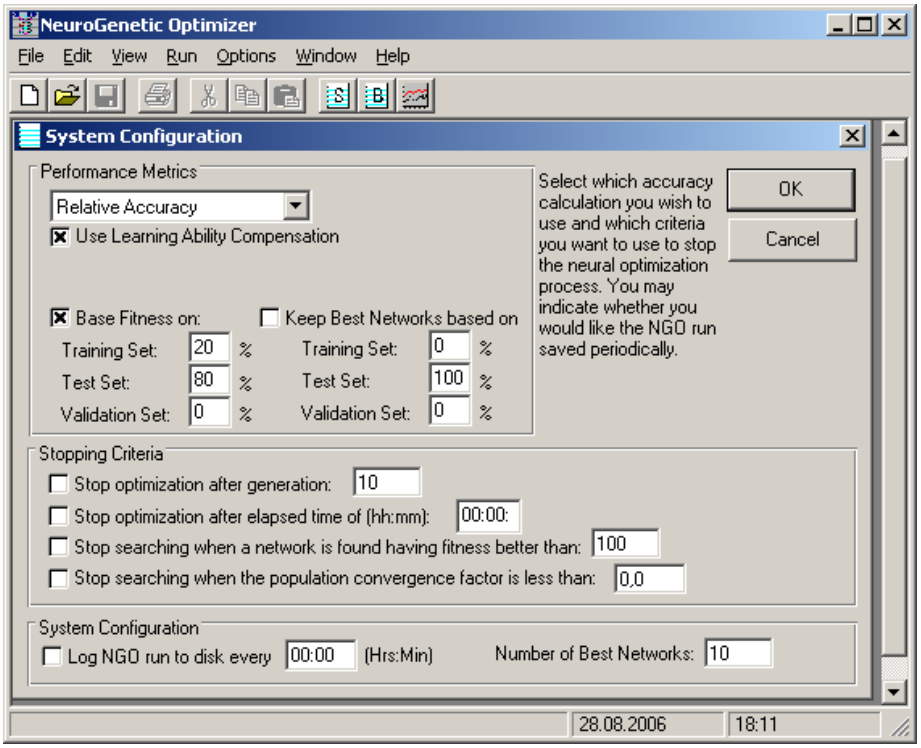
Далее, необходимо настроить параметры нейронной сети и генетического алгоритма (Option / NeuroGenetic…):



Слева вверху нам предлагается выбрать типы нейронных сетей, которые NGO будет пытаться применить к данной задаче. Лучше всего не ограничивать программу в выборе, исключив только два типа сетей для обучения без учителя (SOM и TSOM – самоорганизующиеся карты Кохонена). Следующий важный параметр – число нейронов в скрытых слоях. Значение 8 установлено по умолчанию. Чем больше нейронов, тем выше возможная точность моделирования, но дольше время обучения. Сложность сети должна быть согласована с числом входных векторов данных: Если у нас 50 строк на входе, то бесполезно задавать 256 нейронов.

Справа внизу есть три свойства сети, которые мы можем выбрать как предпочтительные: построить маленькую сеть, использовать как можно меньше входных переменных, использовать все переменные, но оптимизировать структуру сети. Второй пункт следует выбирать, когда число входных переменных очень велико, что будет вызывать проблемы с использованием обученной сети.

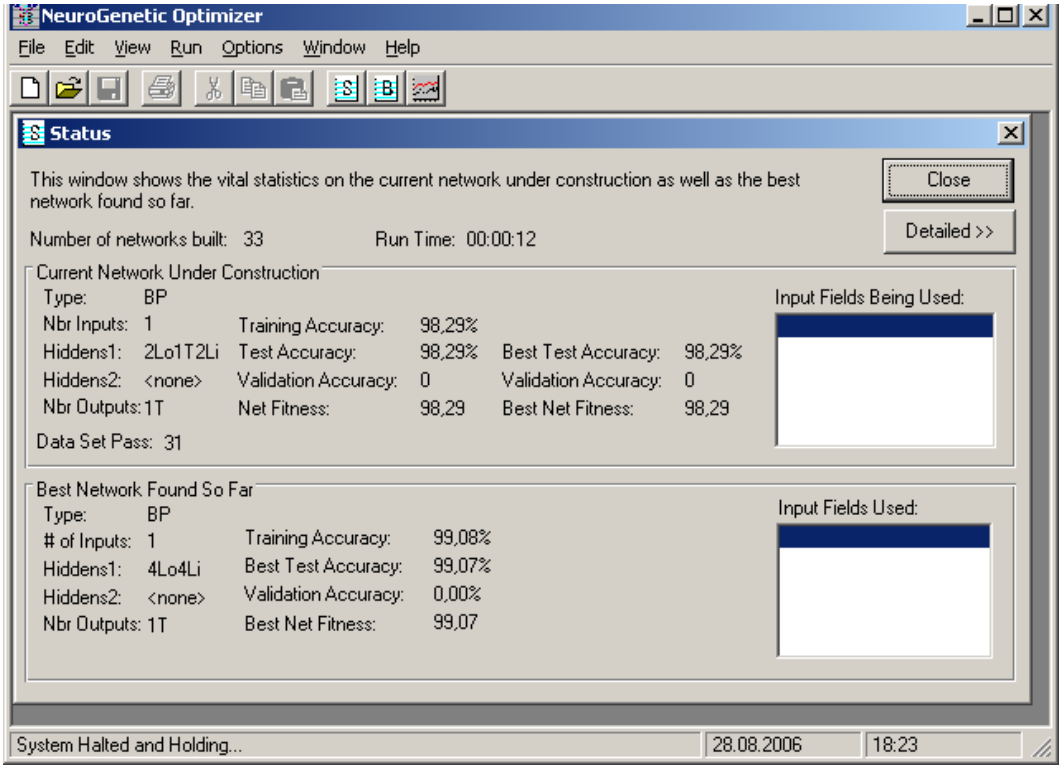
Теперь необходимо установить параметры обучения (Option / System):



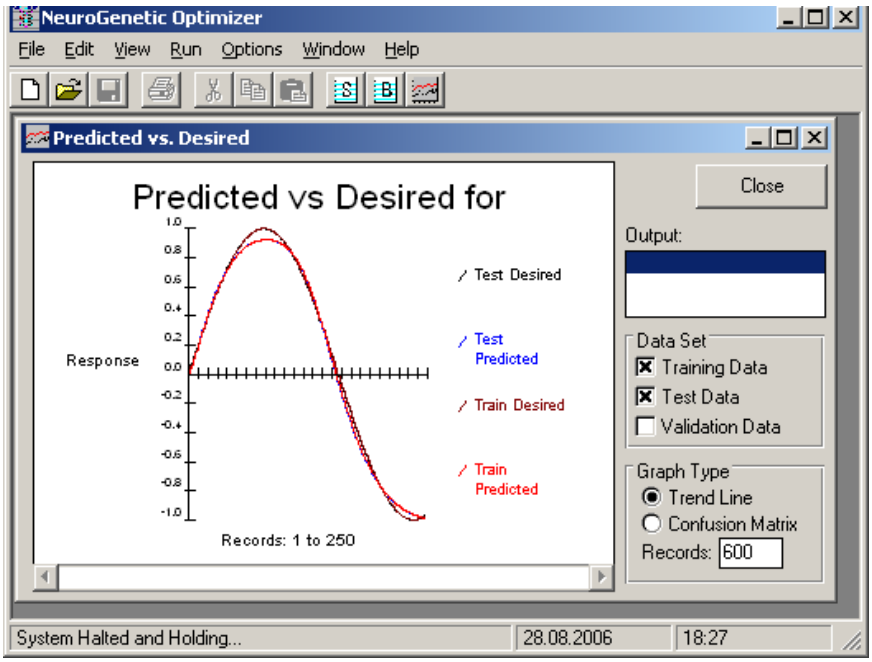
Слева вверху нам предлагается выбрать, как измерять точность. Можно установить абсолютную погрешность, среднеквадратичную ошибку и т.д. Наиболее понятной является относительная точность в процентах (как показано на снимке). Следующая группа параметров определяет, как оценивать точность обученной сети. Точность оценивается параметром Fitness, который учитывает в показанном примере 20% погрешности, полученной на данных обучения и 80% - точность, показанную на этапе тестирования. Иными словами, погрешность на тестировании в 5 раз важнее погрешности обучения.

Следующая группа параметров – критерии завершения обучения. По умолчанию устанавливается завершение после 10 поколений «генетически модифицированных» сетей. Если не устанавливать критерий, то процесс обучения будет бесконечным, и остановить его нужно будет с помощью пункта меню Run / Halt.

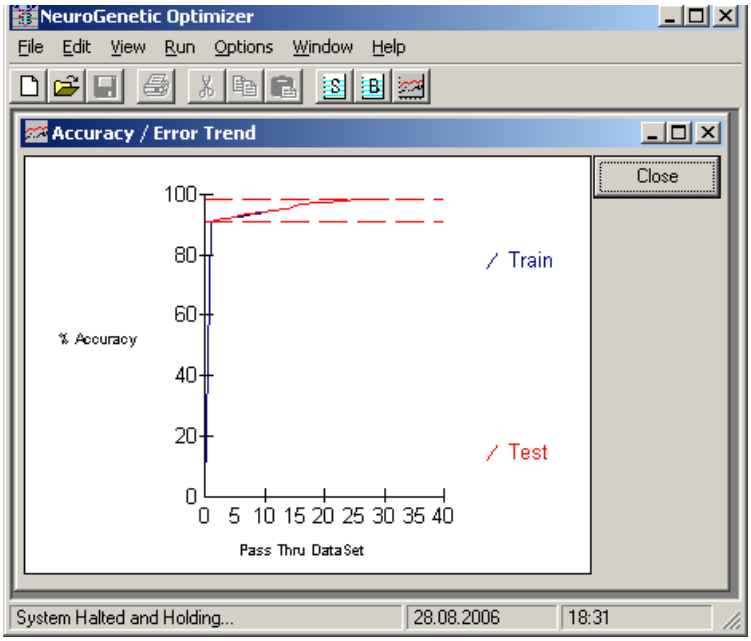
Теперь можно запускать обучение сети (Run / Start). Процесс обучения можно контролировать (View / Status) или с помощью кнопки S на панели инструментов. Появится окно следующего вида, в котором отображаются характеристики текущей конфигурации сети (верхняя панель) и характеристики лучшей сети их всех, сгенерированных на данный момент.



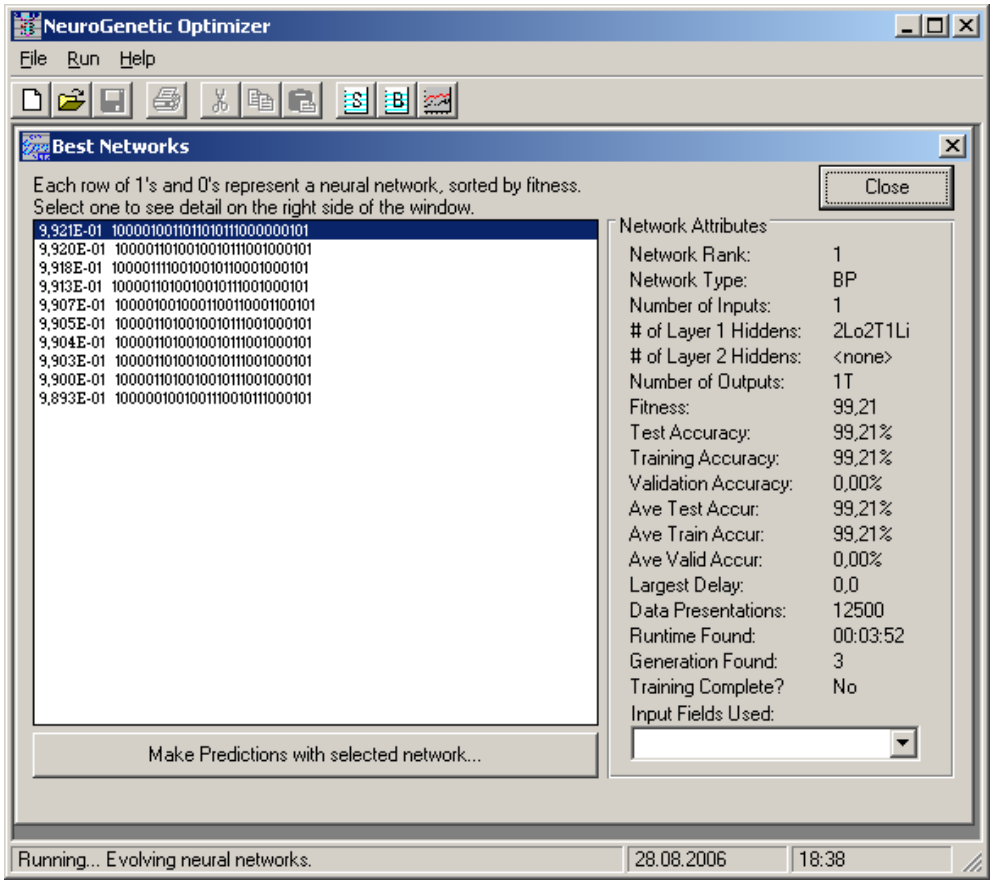
Можно также наблюдать на графике процесс подгонки модели (View / Predicted vs. Desired - предсказанные и желаемые значения),



а также процесс изменения ошибок (View / Accuracy/Error Trend)

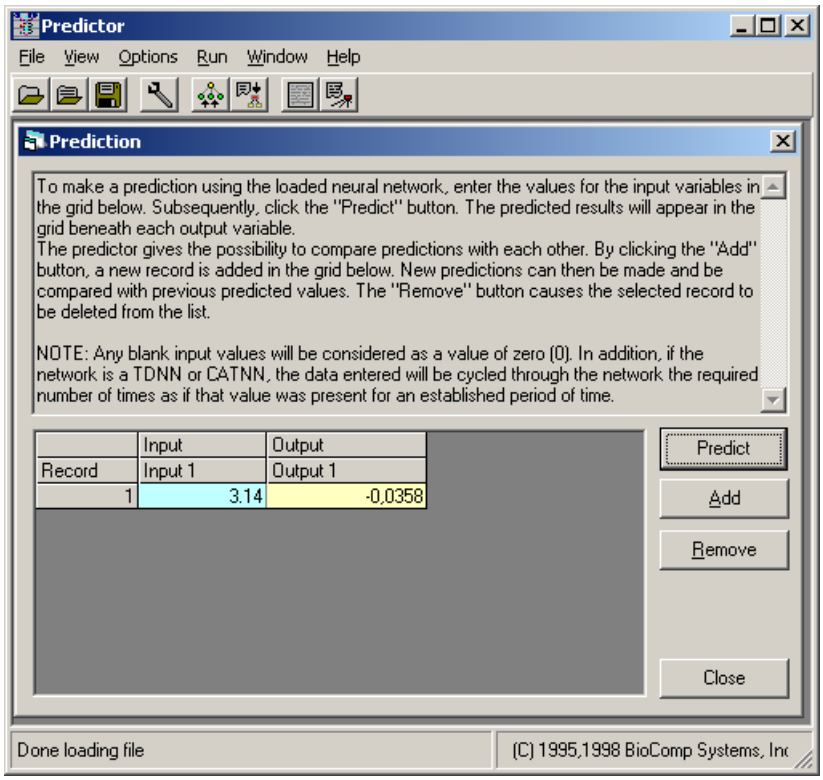


Процесс обучения можно завершить, когда точность модели будет удовлетворительной. Теперь можно приступить к предсказанию. Для этого надо выбрать View / Best Networks или нажать кнопку B на панели инструментов. Появится следующее окно:



Нужно выбрать одну из сетей и нажать кнопку “Make Predictions with selected network…”. Запускается программа Predictor, которая предлагает посмотреть график с результатами обучения, сделать быстрый прогноз путем ввода входных данных с клавиатуры, либо сделать прогноз на основе файла с данными.

Если мы выберем быстрый прогноз, то в появившуюся пустую таблицу нужно ввести входной вектор и нажать кнопку Predict:



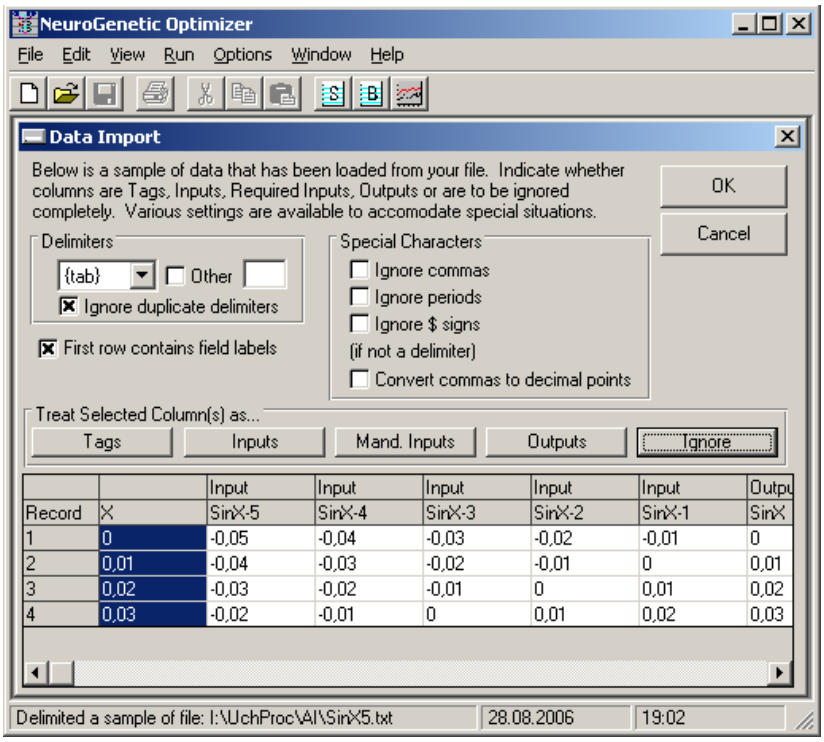
Важное замечание. Числа нужно вводить с десятичной точкой, а не запятой.

Несмотря на достигнутую высокую точность, мы легко можем выявить очевидный недостаток нейронных сетей: они не понимают, что такое синус, а просто запоминают, как нужно отвечать. Если мы введем аргумент 6, то получим результат -1 вместо ожидаемого -0,27942. Нейронная сеть правильно выдает результат только в том диапазоне существования входных данных, на котором она была обучена. Вне этого диапазона результаты непредсказуемы.

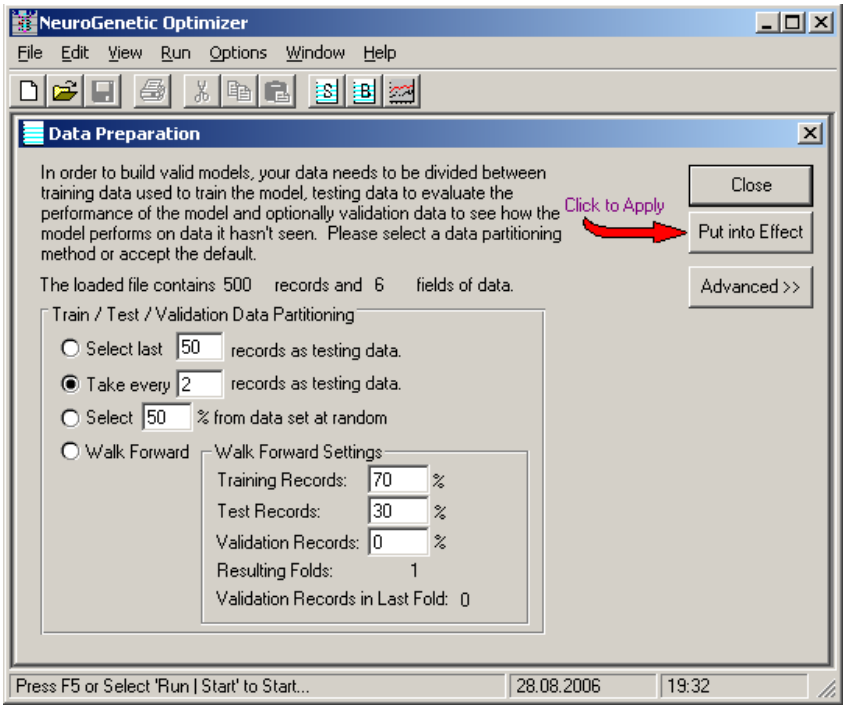
Рассмотрим теперь, как применить программу NGO для прогнозирования значений временных рядов. Пусть та же самая функция синуса представлена не векторами (X, sin(X)), а последовательностью значений Sin(X), и мы хотим прогнозировать значения этого временного ряда на основе наблюдаемых предыдущих значений. Для этого создадим следующий набор данных:

X	SinX-5	SinX-4	SinX-3	SinX-2	SinX-1	SinX
0	-0,050	-0,040	-0,030	-0,020	-0,010	0,000
0,01	-0,040	-0,030	-0,020	-0,010	0,000	0,010
0,02	-0,030	-0,020	-0,010	0,000	0,010	0,020
0,03	-0,020	-0,010	0,000	0,010	0,020	0,030
0,04	-0,010	0,000	0,010	0,020	0,030	0,040
0,05	0,000	0,010	0,020	0,030	0,040	0,050
0,06	0,010	0,020	0,030	0,040	0,050	0,060
0,07	0,020	0,030	0,040	0,050	0,060	0,070

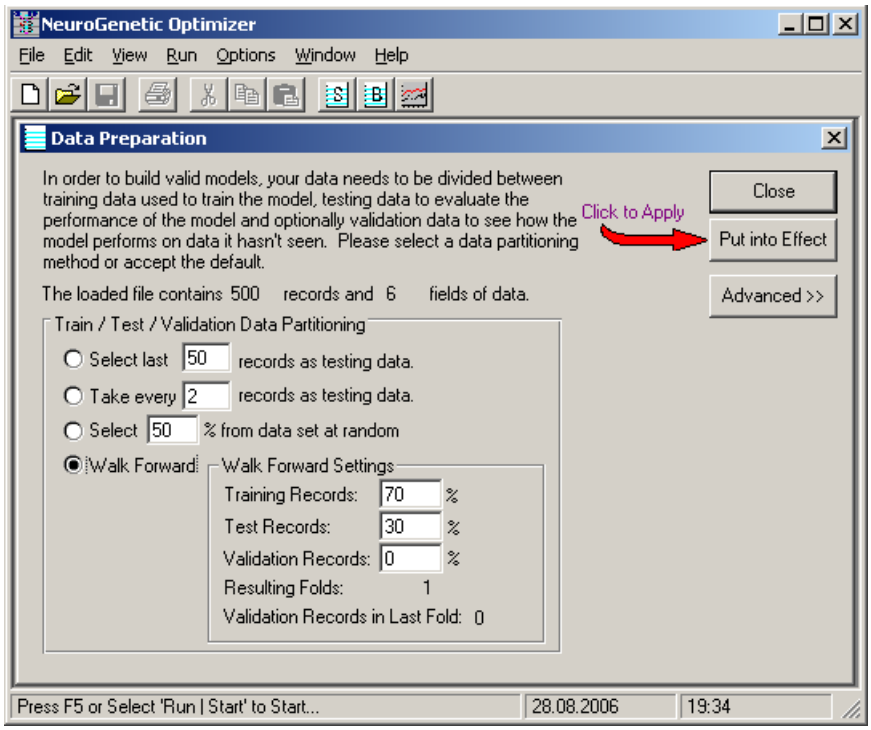
Здесь первый столбец приведен для справки, в обучении он использоваться не будет. Загрузим данный файл в программу NGO:



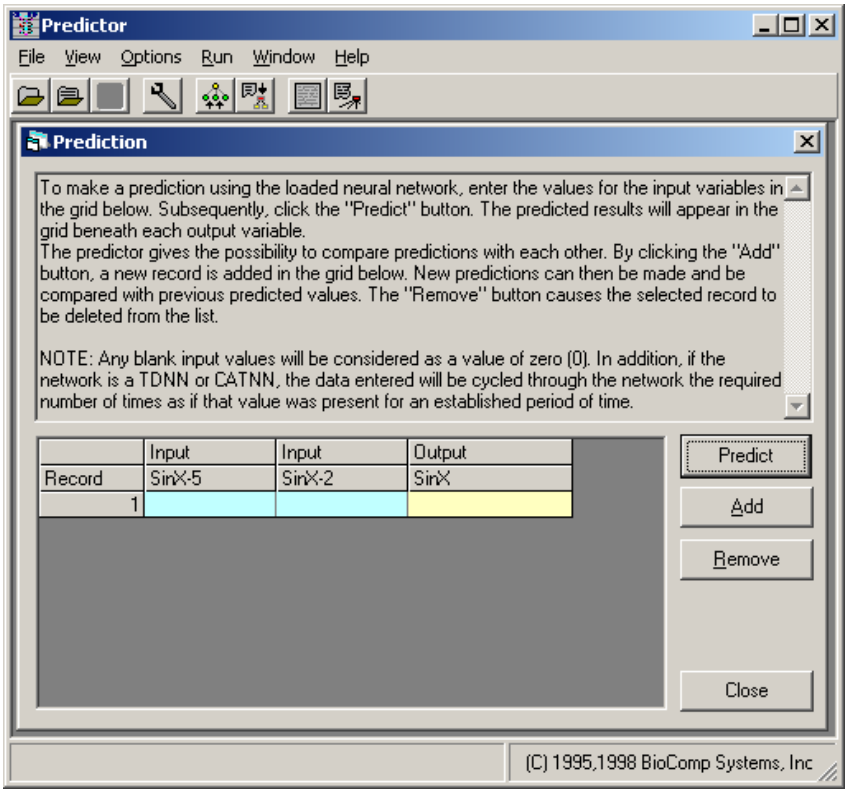
Переменную X помечаем как Ignore, все остальные, кроме последней, как входные. Теперь нужно указать программе, как оперировать с этими данными.



Очевидно, что брать каждый второй вектор, как тестовый, нецелесообразно, так как нам важно сохранить хронологию. Использовать последние записи, как тестовые, также не годится. Здесь целесообразно использовать скольжение вдоль временного ряда: скажем, берем первые 7 строк для обучения, следующие 3 – для тестирования. После этого сдвигаемся на одну позицию вниз: 7 строк, начиная со второй – для обучения, следующие 3 – для тестирования. Сдвигаемся еще на одну позицию вниз и т.д.



Нажимаем кнопку “Put into Effect”, затем – “Close”. После этого устанавливаем остальные параметры, как в предыдущем примере, и запускаем обучение. После того, как точность модели нас станет удовлетворять, останавливаем процесс обучения и запускаем Predictor (View / Best Networks / Make Predictions…).



В выбранной нами сети всего две входных переменных, которые нам надо ввести: SinX-5 и SinX-2, т.е. значения временного ряда, отстоящие от прогнозируемого значения на 5 и на 2 позиции. Введем соответствующие значения из какой-либо строки таблицы и получим результат.

Если нужно делать прогнозы на основе данных, которые доступны путем экспорта из других источников, то необходимо формировать файл такой же структуры, как и набор данных для обучения / тестирования, и использовать опцию “Make predictions from a file of data”. Когда сеть обучена, то в дальнейшем вызывать программу прогнозирования из обучающей программы нет необходимости. Напрямую прогнозирование запускается с помощью файла PRE32150.EXE. Обученная сеть должна быть сохранена в обучающей программе пунктом меню “File / Save Network”.

Приложение 3. Программа Semantic. Руководство пользователя

1 Назначение и условия применения программы

Программа предназначена для представления и визуализации знаний в виде семантических сетей, а также для доступа к базам знаний с помощью графического интерфейса и языка запросов.

Программа может использоваться в режиме обучающей системы, в котором база знаний содержит в неявном виде некоторую гипотезу, например, диагноз, а пользователь, задавая вопросы, должен выявить данную гипотезу, затратив на это минимум вопросов. Кроме того, программа может работать в режиме экспертной системы, сопоставляя имеющиеся факты с правилами в базе знаний.

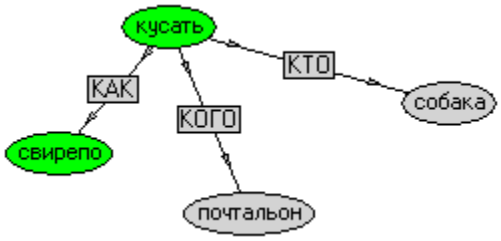
Для выполнения программы требуется персональный компьютер под операционной системой Windows XP. Особых требований к ресурсам не предъявляется. Для хранения программы необходимо не менее 16Мб на жестком диске.

Программа разработана для обучения студентов основам построения семантических сетей в рамках дисциплины «Искусственный интеллект». Программа написана на языке ***VISUAL PROLOG V 7.2 Personal Edition*** (www.pdc.dk). Данная программа предназначена исключительно для образовательных целей. Использование данной программы в коммерческих целях не допускается.

2 Характеристики программы

Программа обеспечивает создание и использование семантических сетей, как один из способов представления знаний. В настоящей версии программы поддерживаются сети с бинарными отношениями, т.е. связывающие ровно два понятия, одно из которых является субъектом, второе – объектом. Триплет субъект-предикат-объект образует факт.

Одно и то же понятие может присутствовать в нескольких фактах, что и обуславливает сетевую структуру. Программа допускает также создание семантических сетей на основе графов Растье (графов с глаголом в центре), а также комбинация обычных реляционных графов с графами Растье. Граф Растье содержит сведения не об отношениях между субъектами и объектами, а о процессах (действиях). Ниже приведен часто используемый пример описания с помощью графа Растье следующего события: Собака свирепо покусала почтальона. Сеть построена вокруг глагола КУСАТЬ. Агентом (субъектом) кусания является собака, объектом – почтальон. СВИРЕПО – манера кусания.



Преимущества графа Растье заключается в возможности установления свойств, относящихся не к объекту или субъекту, а к процессу. СВИРЕПО – это не свойство собаки, а свойство процесса кусания почтальона.

Программа позволяет создавать семантические сети на разных языках, в том числе многоязычные сети. Текстовый интерфейс пользователя позволяет извлекать факты из базы знаний на упрощенном естественном языке. Поддерживается модульная структура представления знаний, т.е. размещение разных предметных областей в разных файлах.

Визуализация знаний обеспечивается с помощью графического представления семантической сети. Графический пользовательский интерфейс позволяет развертывать граф в глубину и в ширину, а также откатываться назад, менять тему диалога, т.е. управлять контекстом. Под контекстом здесь и далее понимается набор фактов, уже извлеченных из базы знаний.

Для упрощения отладки создаваемых семантических сетей все действия программы протоколируются с помощью сообщений в окне “Messages”.

3 Входные данные

Базы знаний семантических сетей хранятся в текстовых файлах и содержат знания в синтаксисе языка Prolog, поскольку загружаются в программу как база фактов предикатом consult. Одна база знаний соответствует одной предметной области. На имена файлов не накладываются ограничения. Редактировать файла баз данных можно с помощью любого текстового редактора. Однако, если использовать редактор в составе компилятора Prolog, то это позволит обнаруживать ошибки, используя возможности среды отладки Prolog. Тогда для редактора в составе SWI-Prolog файлы целесообразно снабжать расширением “.pl”, а для Visual Prolog – “.pro”.

Каждый файл базы знаний, описывающий предметную область, может содержать ссылки на файлы онтологий, в которых хранятся словари терминов и правила, с помощью которых из фактов могут извлекаться новые факты. Кроме того, файл базы знаний может содержать ссылки на другие файлы баз знаний, т.е. другие предметные области. В данной редакции в целях упрощения написания путей все файлы, используемые программой, должны храниться в одном и том же каталоге.

В пределах одной предметной области должна соблюдаться уникальность именования понятий. С увеличением числа фактов эта проблема может

привести к усложнению добавления новых фактов, поскольку нужно соблюдать уникальность имен, и ухудшению читаемости знаний. Если база знаний становится слишком громоздкой, целесообразно разбить файл на несколько баз знаний, которые размещаются в отдельных файлах. Каждый файл соответствует отдельной теме диалога. Данный прием позволяет переключаться с одной темы диалога на другую аналогично тому, как это делается в реальном разговоре. Такое переключение будем называть сменой контекста.

3.1. Файл базы знаний

Файл базы знаний имеет произвольное имя. В этом файле могут размещаться следующие записи:

Имя	Назначение	Формат	Пример
t	Наименование основной темы	t(<субъект>).	t(″Jack″).
pic	Имя стартового рисунка в формате BMP. Необязательный.	pic(<имя файла>).	pic(″mailman.bmp″).
onto	Имя файла онтологии.	onto(<имя файла>).	onto(″common_eng.pro″).
e	Описание внешней ссылки	e(<субъект>,<имя файла>).	e(″engine″, ″engine.txt″).
f	Описание факта	f(<субъект>, <предикат>, <объект>).	f(″заяц″, ″есть″, ″млекопитающее″).
hypo	Описание гипотезы	hypo(<субъект>).	hypo(″anaemia″).

Ниже приведены пояснения к данным описаниям.

Все записи в файлах баз знаний подчиняются синтаксису предикатов языка Prolog. Каждое выражение начинается имени, начинающегося со строчной латинской буквы. В скобках через запятую размещаются аргументы. Выражение заканчивается точкой.

Предикат описания факта **f** – основной элемент базы знаний. Все элементы триплета должны заключаться в кавычки (знак ″, но не ” или “). Не допускается использовать в них пробелы и символы-разделители, если предполагается их

использовать в запросах, поскольку парсинг запроса осуществляется как разбор текста на слова. Пробелы между символами-разделителями допускаются: f(″заяц″ , ″один_из″ , ″млекопитающее″).

3.2. Файл онтологий

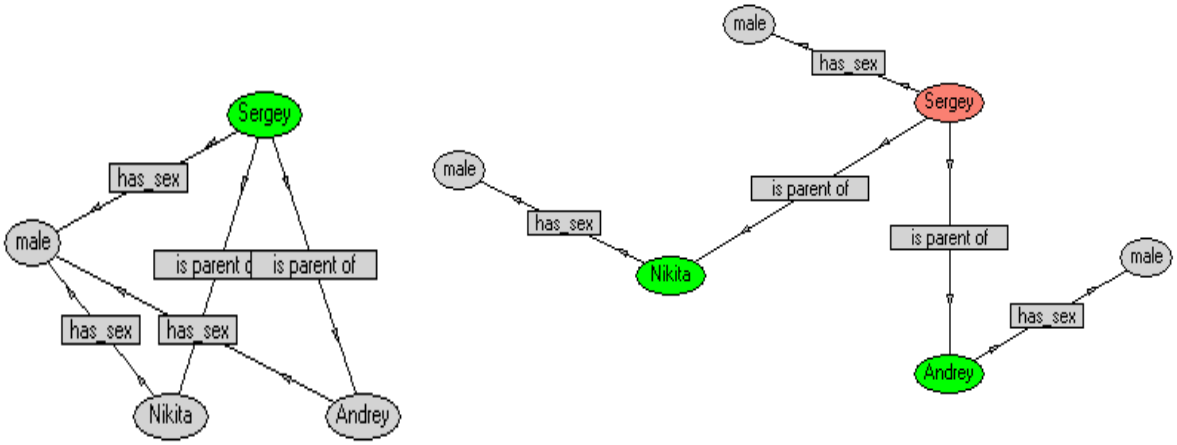
В файле онтологий могут также располагаться записи следующих типов.

Имя	Назначение	Формат	Пример
c	Описание класса	c(<слово>).	c("person"). c("everybody"). c("thing").
o	Факт. То, же, что и f(…) в файле базы знаний.	o([<субъект>, <предикат>, <объект>]).	d(["everybody","a_kind_of", "thing"]).
d	Описание объектов и субъектов (dictionary)	d([<слово>, <слово>,…, <слово>]).	d(["car","автомобиль", "voiture"]).
p	Описание предикатов (predicates)	p([<слово>, <слово>,…, <слово >]).	p(["ERG","КТО","Агент"]).
pr	Описание свойств (properties)	p([<слово>, <слово>,…, <слово >]).	p(["has_name"]). p(["мужской”, “male"]).
g	Предпочтительная ориентация связи для лучшей читаемости графа	g(<направление>, [<слово>,<слово> ,…, <слово >]).	g(down,["is_parent", ”is_grandparent", "is_uncle", "is_father"]). g(side,["is_cousin", "is_sibling"]). g(up, ["is_a", "is_child"]).
r	Предикат описания правила	r(<причина>),<следствие>),	r([t("?x", "брат", "?y")], [t("?x", "родственник", "?y")]).

Описание класса **c** необходимо для того, чтобы отличить класс от экземпляра.

Запись типа **o** используется точно так же, как и запись **f** в файле базы знаний. Необходимость присвоения другого имени обусловлено ограничениями языка Пролог. Поскольку файл онтологий предназначен для хранения знаний на уровне классов, а не экземпляров, запись типа **o** описывает отношения между классами с свойства классов.

Если в составе семантической сети используются синонимы терминов, то их необходимо описать в записях типов **d** и **p**. Запись типа **d** описывает синонимы объектов и субъектов (dictionary), а запись типа **p** – синонимы предикатов (predicate). Записи типа **pr** необходимо создавать даже при отсутствии синонимов, поскольку граф для свойств строится иначе, чем для отношений. Если не указать свойство в записи **pr**, то одинаковые свойства всех объектов будут сводиться в одну вершину, например, так, как показано ниже. На левом рисунке свойства сводятся в одну вершину, на правом – нет.



Кроме того, идентификация свойств полезна для фильтрации графа, если граф становится слишком плотным, целесообразно исключить отображение на нем свойств, оставив только отношения. Запись типа **pr** может включать как наименование свойства, так и значение, например, “has_sex” или “male” и “female”.

Все синонимы одного термина должны размещаться в одном списке. Использование словарей позволяет сравнительно просто сделать семантическую сеть независимой от языка. При этом первый элемент каждого списка будет всегда использоваться для отображения на графе.

Внимание:

1. Если в онтологии присутствует запись о синонимах типа d или p, программа везде будет отображать первый элемент списка в качестве идентификатора понятия. Это свойство программы можно использовать для создания многоязычных знаний. Достаточно создать онтологию с записями вида p([“один_из”,”ISA”]), и отображаться будут только русскоязычные термы.

2. В данной версии программы допускается только один список синонимов для каждого понятия. Если в разных онтологиях, подключаемых к базе знаний, будет присутствовать запись типа d или p для данного понятия, программа использует только один из них. Например, в одном файле онтологии указано p(["ISA", “is_a”]), а в другом – p([“один_из”,”ISA”]), то программа будет использовать только один из списков в зависимости от порядка следования записей onto d файле базы знаний.

3.3. Правила в базе знаний

Правила позволяют устанавливать закономерности, на основе которых можно получать новые знания. В правилах используются переменные, которым в процессе унификации с фактами присваиваются значения. В связи с тем, что внешние файлы Пролога не допускают использование переменных, здесь принято специальное их именование. Переменные должны быть представлены текстовыми константами, начинающимися с вопросительного знака, например, ″?x″, ″?class″. Правило состоит из двух списков триплетов. Второй список содержит новые знания, которые становятся истинными, если действительны факты из первого списка. Пример:

r([t(″?x″, ″родитель″ , ″?y″), t(″?y″, ″родитель″ , ″?z″), t(″?z″, ″пол″ , ″мужской″)], t(″?z″, ″внук″ , ″?x″)]).

Данная запись эквивалентна правилу на Прологе

grandchild(X, Y) :- parent(Y, Z), parent(Z, X), sex(X, male).

Несмотря на то, что программа Semantic написана на Прологе, использовать такое правило напрямую невозможно, поскольку записать его можно только в текст программы, а не во внешнюю базу знаний. Во внешней базе Пролога допускается хранение только фактов.

В случаях, когда требуется установить идентичность / различие двух объектов, в правиле должен использоваться специальный предикат “differs”. Приведенный ниже пример показывает правило для отношений брат или сестра:

r([t("?x","is_parent","?y"),t("?x","is_parent","?z"),t("?y","differs","?z")],
[t("?y","is_sibling","?z")]).

Если в правиле требуется отрицание, то используется конструкция n(<субъект>,<предикат>,<объект>).

Эта конструкция аналогична отрицанию отношения t(<субъект>,<предикат>,<объект>).

Ниже показан пример правила, описывающего отношение отчим/мачеха –> пасынок/падчерица.

r([t("?x","is_parent","?y"),t("?x","is_spouse","?z"), n("?z", "is_parent", "?y")],
[t("?z","is_step-parent","?y")]).

Большее количество примеров правил содержится в папке Examples.

3.4. Правила наследования

В программе реализованы правила наследования отношений, которые автоматически применяются при каждом обращении к базе знаний после того, как все факты исчерпаны. Применяются следующие правила наследования:

Если X AKO Y и Y AKO Z то X AKO Z
Если X ISA Y и Y AKO Z то X ISA Z
Если X has_part Y и Y has_part Z то X has_part Z
Если X ISA Y и Y has_part Z то X has_part Z
Если X AKO Y и Y has_part Z то X has_part Z
Если X ISA Y и Y has_a Z то X has_a Z
Если X AKO Y и Y has_a Z то X has_a Z

Для того, чтобы указанные правила применялись, отношения ISA, AKO и т.д. должны указываться в триплетах фактов именно так, как здесь, либо следует создавать синонимы с помощью записей типа

p([“is_a”, “ISA”]).
p([“a_kind_of”, “AKO”]).

3.5. Принципы идентификации объектов

Используемые в фактах обозначения объектов являются их локальными идентификаторами, которые действуют только в пределах одного файла баз знаний. Например, факт f(“Sergey”,”is_parent”,”Nikita”) вовсе не обязательно относится с семейству Михалковых, даже если рядом присутствуют факты f(“Sergey”,”is_parent”,”Andrey”), f(“Andrey”,”is_parent”,”Egor”), которые у всех на слуху. Полностью идентифицируют объект такие его свойства как “has_name”, “has_surname”, “has_birth_date” и т.п., включая номер паспорта. Однако, такая строгая идентификация сильно загромоздит запросы, да и не требуется в учебных целях. Мы будем считать, что идентификация предметной области содержится в комментариях в самом тексте файла, которые доступны пользователю.

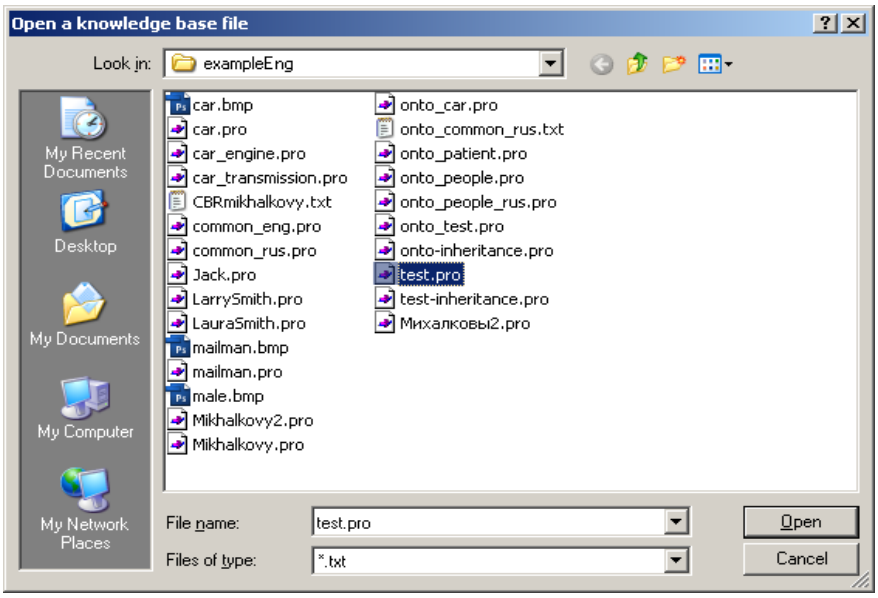
Программа требует уникальной идентификации вершин графа, и, следовательно, уникальных идентификаторов объектов в пределах одного файла. В графах Растье соблюсти уникальность сложно, поскольку в центре ставиться глагол (процесс), разнообразие которых существенно меньше, чем объектов. Здесь допускается присвоение уникального идентификатора перед именем процесса. Например, в одном графе используются два глагола «жить», относящиеся к разным объектам. В этом случае вместо «жить1» и «жить2» можно записать «1:жить» и «2:жить». Префиксы перед двоеточием на графе отображаться не будут. Кроме того, в результате применения правил могут появляться объекты с одинаковыми названиями. Пусть в базе онтологий есть факт “person has_part hand” (у человека есть рука). После применения правил наследования к нескольким экземплярам могут появиться одинаковые идентификаторы: “Ivan has_part arm”, “Stepan has_part arm” и т.д. На графе, да и не только это может воспринято как факт, что все эти люди имеют одну и туже составную часть (сиамские близнецы). Для устранения неоднозначностей программа автоматически подставляет к наименованию объекта префикс – имя субъекта: “Ivan has_part Ivan:arm”, “Stepan has_part Stepan:arm”. Такая

идентификация позволяет избежать двусмысленностей; при этом на графе префиксы не отображаются, поскольку расположение связей позволяет идентифицировать объекты.

4 Запуск программы

Каталог программы должен хранить следующие файлы, требуемые для ее выполнения: Semantic.exe, Vip6u2a.dll, Vip7edit.dll, Vip7kernel.dll, Vip7regexp2.dll, Vip7run.dll, Vip7vpi.dll. Исполняемый модуль – Semantic.exe.

Сразу после старта программа выдает диалоговое окно для выбора файла глобальной базы знаний, которое выглядит следующим образом:

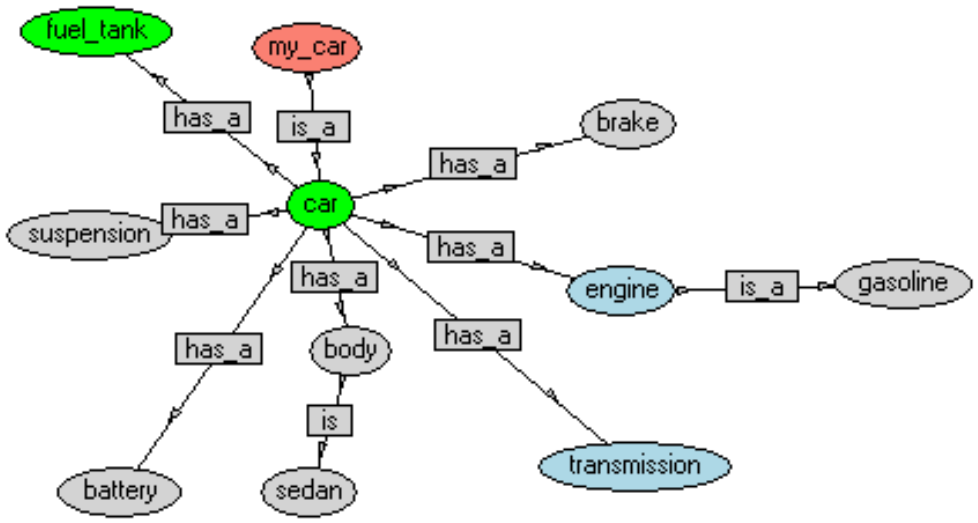


После того, как файл глобальной базы знаний открыт, появляется основное окно программы, которое описано в разд. 7 «Интерфейс программы».

5 Интерфейс программы

5.1. Граф семантической сети

Граф семантической сети, отображаемый с помощью программы, выглядит следующим образом:



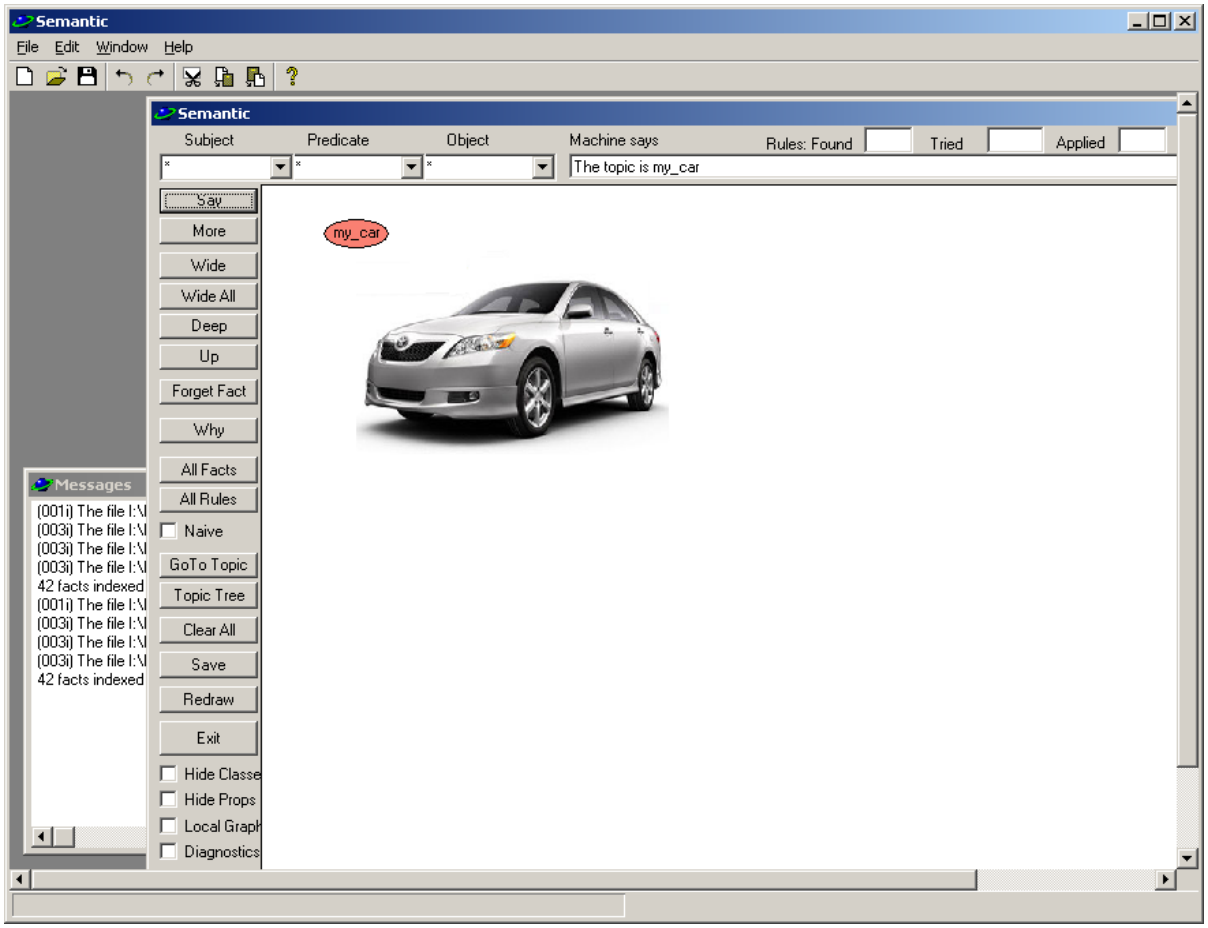
Субъекты и объекты отображаются в виде эллипсов с надписями, а отношения (предикаты) – прямоугольниками с надписями. Для наглядности графа не рекомендуется давать субъектам, объектам и предикатам длинные названия, иначе фигуры будут накладываться друг на друга. Приняты следующие цветовые обозначения:

1. Розовым цветом обозначена основная тема.
2. Желтым цветом выделяются классы.
3. Красным цветом обозначается факт или отдельный объект или субъект, выбранный мышью.
4. Голубым цветом обозначается объект или субъект имеющий ссылки на внешний файл.
5. Последний факт, включенный в контекст, окрашен в зеленый цвет.
6. Оранжевым цветом обозначаются причинно-следственные связи.

В программе не в полной мере реализована перерисовка основного окна. Если закрыть его другим окном, изображение не восстанавливается. Чтобы восстановить граф, можно нажать кнопку **Redraw** или изменить размер или положение окна на экране. Пока не поддерживается также скроллинг окна.

5.2. Элементы управления

После запуска программы и загрузки файлов с базами знаний на экране отображается основное окно программы, которое выглядит следующим образом:



В окне программы имеется основная панель **Semantic** и окно сообщений **Messages**.

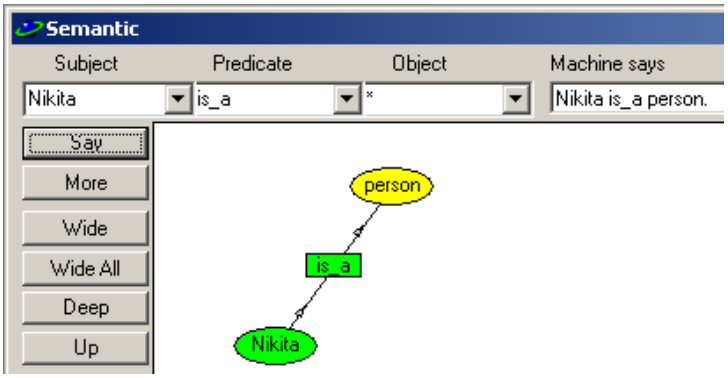
На панели “Semantic” имеются три поля для ввода запроса в виде Субъект-Объект-Предикат. Для каждого из этих полей предусмотрены выпадающие для выбора значения из существующих фактов и правил. Рядом с ними находится поле диалога (“Machine says”). Поле “Questions counter” содержит счетчик запросов к базе знаний и может использоваться для контроля знаний в режиме обучающей системы. Кроме счетчика вопросов имеются счетчики правил: Число найденных правил (**Rules Found**), число попыток их применения (**Tried**) и число успешных попыток (**Applied**).

Единственная действующая опция основного меню программы – “File/New” или кнопка “New” – позволяет открыть новое окно программы без ее перезапуска. Параллельно можно открывать неограниченное число окон. Ограничение связано только с объемом доступной оперативной памяти.

Кнопки основного окна имеют следующее назначение:

Say – Обработать триплет, введенный в поле для ввода.

Например, на запрос “Nikita is_a *”, получаем ответ: “Nikita is_a person” и соответствующий фрагмент семантической сети.

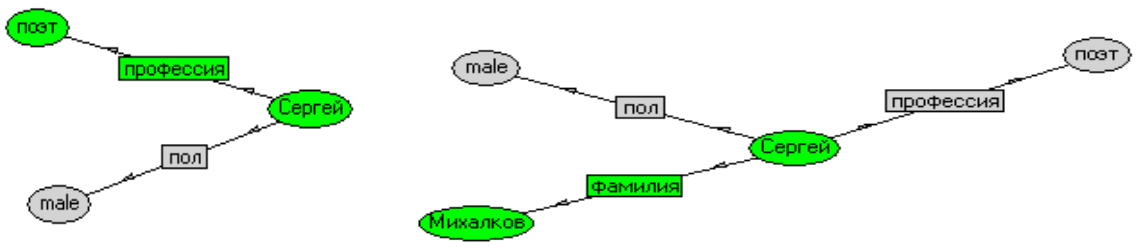


Вместо конкретного значения в любом поле ввода можно оставить звездочку. Это будет означать, что нас интересуют факты с любым значением в данном поле.

Если фактов, релевантных данному запросу, в базе больше нет, программа применяет цепочку обратного вывода (Back Chain Reasoning). Это может занять продолжительное время.

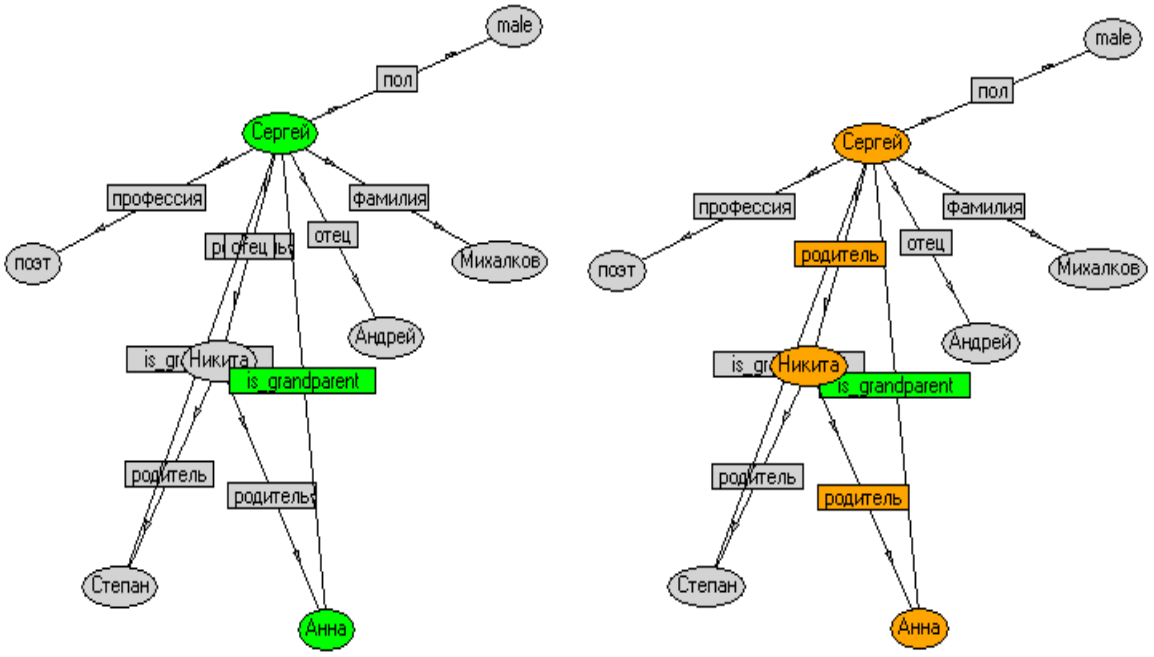
More – получить альтернативный ответ на запрос по кнопке **Say**. Например, на вопрос “* has_child Nikita” программа дает ответ: “Sergey1913 has_child Nikita”. Нажатие кнопки More позволяет получить альтернативный ответ: “Natalia1903 has_child Nikita”, как показано ниже. Так же, как и по кнопке **Say** при отсутствии релевантных фактов запускается обратный вывод.

Wide – получить факт о другом объекте, связанным с выбранным субъектом или субъектом, полученном в последнем обращении, т.е. получить соседнюю ветвь по отношению к последней. (Последний факт, включенный в контекст, отображается на графе зеленым цветом, а объект, выбранный двойным щелчком мыши – красным цветом). Например, последний факт был “Сергей профессия поэт”. Если нажать кнопку **Wide** еще раз, то будет получен факт “Сергей фамилия Михалков” и т.д.



При каждом нажатии кнопки **Wide** будет извлекаться очередной факт из базы знаний и пополнять контекст. После того, как все факты, относящиеся к данному субъекту, будут извлечены, программа начнет пытаться применить правила. Ниже показано, что Сергей является прародителем Анны. Если нажать кнопку **Why**, то оранжевым цветом будут выделены факты, являющиеся

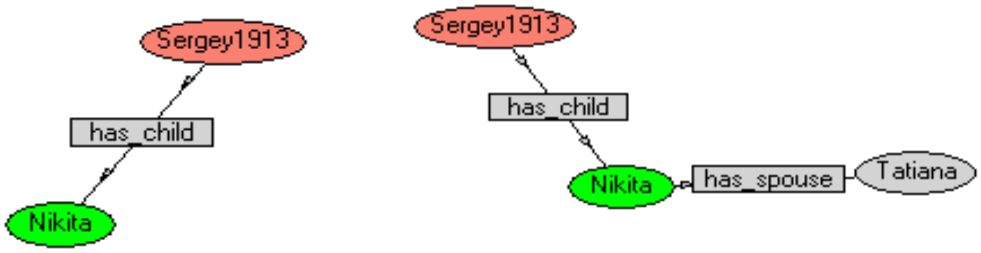
основанием для сделанного вывода,



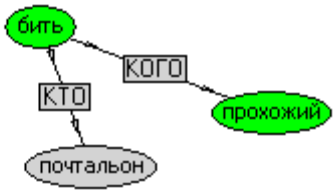
а в поле диалога и окне сообщений программы - текстовое объяснение: **Сергей is_grandparent Анна ПОТОМУ ЧТО Сергей родитель Никита И Никита родитель Анна.**

Кнопка **Wide All** находит все объекты, связанные с данным субъектом, в том числе пытается применить все правила.

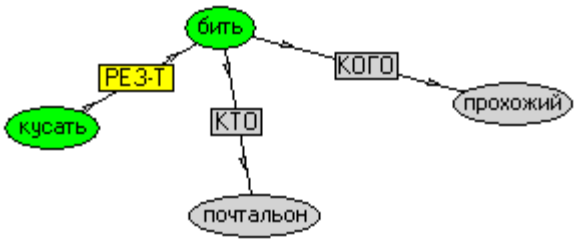
Deep – получить следующий факт об выбранном объекте или объекте, полученном в последнем обращении, т.е. углубиться по текущей ветви графа. Например, последний факт был “Sergey1913 has_child Nikita.”. При углублении по данной ветви получим факт “Nikita has_spouse Tatiana.”.



Up – получить факт, в котором последний или выбранный субъект является объектом, т.е. подняться выше по графу.



Если сейчас нажать на кнопку **Up**, то мы получим другой субъект, связанный с объектом “бить”:



Forget Fact – удалить из контекста факт, объект которого предварительно выделен красным цветом с помощью двойного щелчка мыши.

All Facts – найти и отобразить все факты. По нажатию этой кнопки программа включает в контекст все факты из базы знаний, а также пытается применить ко всем этим фактам правила наследования добавляет в текущий контекст новые факты, установленные на основе этих правил.

All Rules – Применить к базе знаний все известные правила. По нажатию этой кнопки программа последовательно применяет каждое правило ко всем фактам базы знаний. Выполнение этой функции может занять много времени. Если в базе онтологий имеется 10 правил по два триплета в каждом, а в базе знаний – 100 фактов, то количество попыток применения этих правил будет лежать в диапазоне 10*100 – 10*100*100, т.е. от 1000 до 100000. Обработка 15000 попыток применения правил требует около 5 минут времени.

В целях ускорения логического вывода в программе реализован алгоритм индексации фактов, сокращающий время обработки правил приблизительно на два порядка. Для исследования временных характеристик алгоритма в программе также предусмотрен режим без индексации – наивный вывод (переключатель **Naïve**), а также счетчики показывающие значения числа найденных правил (**Rules Found**), попыток применения (**Tried**) и успешных попыток (**Applied**), т.е. правил, выдавших релевантные факты. Счетчики находятся в верхней части экрана. При вызове правил кнопкой **Wide** счетчики показываю нарастающие значения, а по кнопке **All Rules** счетчики предварительно сбрасываются.

Следует заметить, что один цикл применения правил далеко не всегда дает все возможные факты, поскольку многие факты являются результатом выполнения цепочек фактов. Логика программы реализует прямой вывод (forward chaining), т.е. от известных фактов к цели. При этом перебираются все правила, в теле которых есть триплеты, унифицируемые известными фактами. Поиск завершается, когда находится факт, релевантный запросу. Альтернативой данному подходу является обратный вывод (back chaining), в котором результирующая часть каждого правила унифицируется с заданной целью. Обратный вывод в данной редакции программы реализован по кнопкам

Wide, **Ask** и **More**.

Кнопка **All Rules**, на первый взгляд, реализует спуск по дереву поиска на один уровень. Однако, это не совсем так. Каждый найденный факт немедленно включается в состав фактов, используемых следующим правилом, а общее количество нажатий этой кнопки для гарантированного извлечения всех возможных фактов в значительной степени зависит от порядка следования правил в онтологиях. Если первыми стоят факты, результаты обработки которых используются следующими фактами, то все факты могут быть получены за один проход.

Goto Topic – перейти к субъекту или объекту, выделенному красным с помощью двойного щелчка мыши. При этом накопленный контекст «забывается», но сохраняется для последующего восстановления, а выделенный субъект или объект становится основной темой, от которой начинает развертываться граф.

Topic Tree – показать дерево тем, которые были основными, и порядок перехода от темы к теме. На дереве тем можно мышью выделить любую тему и прейти к ней кнопкой **Goto Topic**. При этом сохраненный ранее контекст восстанавливается.

Redraw – Перерисовать граф. Целесообразно использовать эту кнопку, если узлы сети расположились неудачно.

Clear All – очистить контекст и начать все сначала.

Save – сохранить текущую базу знаний. Программа предлагает выбрать имя файла. Сохранять файл целесообразно в случае, если база знаний пополнялась с помощью кнопки **Say**. Кроме фактов базы знаний кнопка Say вызывает сохранение в файл текущего контекста, в т.ч. фактов, полученных с помощью правил. Таким образом, базу знаний можно обновить так, чтобы в следующий раз не запускать длительный процесс применения правил.

Exit – выход из задачи. Опцией меню File / New можно начать работу заново, без перезапуска программы.

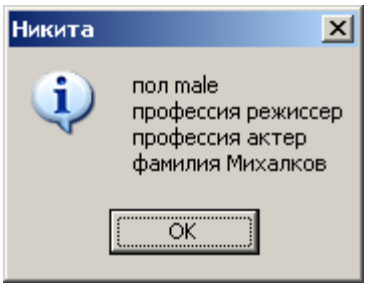
Кроме кнопок имеется переключатели: **Hide Classes**, **Hide Props**, **Local Graph** и **Diagnostics**.

Первые два переключателя позволяют скрыть на графе классы или свойства объектов соответственно, чтобы проредить слишком перегруженный граф. Переключатель **Local Graph** позволяет ограничить граф выбранным объектом и отношениями только с этим объектом. Переключатель **Diagnostics** включает выдачу диагностических сообщений в окне сообщений.

Обработка кликов мышью имеет следующие варианты. Если кликнуть мышью на эллипс, то он подсвечивается красным цветом и в дальнейшем может использоваться для установления новых или удаления установленных фактов (кнопки **Deep**, **Wide**, **Wide All**, **Up**, **Why**), а также перехода к другим темам (**Goto Topic**). Если сделан клик мышью на предикат, то весь факт подсвечивается зеленым цветом и приобретает статус последнего установленного факта. Двойной клик на эллипс вызывает попытку перехода к

новой базе знаний, ссылка на которую должна содержаться в данном объекте (запись типа **e**).

В том случае, если установлен фильтр отображения свойств (**Hide Props**) одиночный клик мышью на объект вызывает еще и выдачу окна со списком свойств данного объекта, как показано ниже.



6 Управление контекстом

По мере увеличения сложности создаваемой семантической сети возникают проблемы. Первая проблема связана со сложностью разрастающегося графа. Заметим, что на экране отображается не весь граф сети, а только та его часть, которая уже принимала участие в диалоге, которую мы будем называть контекстом. По мере вовлечения в диалог новых фактов граф может стать нечитаемым.

Вторая проблема заключается в том, что становится сложнее присваивать уникальные имена понятиям. В пределах контекста одинаковые имена соответствуют одним и тем же понятиям. Здесь существуют две проблемы: полисемия и синонимия. Полисемия – несколько значений одного и того же слова. Например, поле – это аграрное понятие для фермера, физическое для инженера, и правовое для юриста. Полисемия может привести к тому, что фрагменты семантической сети будут объединяться совсем не так, как ожидается. Синонимия – несколько терминов для одного и того же понятия, например, вертолет и геликоптер. Вследствие синонимии наоборот, фрагменты сети будут несвязными. Проблема синонимии затрудняет также объединение фрагментов сети, написанных разными авторами.

В реальном диалоге эта проблема решается на уровне контекста, который оба собеседника подразумевают одинаковым. В случаях, когда участники диалога имеют в виду различный контекст, обычно имеют место недоразумения. Например, вы назначаете встречу у метро на обычном месте. Для вас это место – на выходе из метро, а собеседник думает, что это место у поездов. Не разминуться будет невозможно.

Управление контекстом позволяет привести диалог к виду, привычному для человека. Во-первых, на экране отображается текущий контекст, что позволяет быть в курсе того, о чем идет речь, если пользователь и забыл. В

частности, отображаются текущие значения Во-вторых, меняя контекст, можно временно «забыть» о текущей теме, начав диалог заново. После этого можно снова вернуться к обсуждаемой теме, предложив компьютеру «вспомнить» то, о чем говорилось ранее. Для этого используются возможности программы, реализованные с помощью кнопок **Goto Topic**, **Forget Fact**, **Topic Tree**.

При отображении понятий, для которых имеется отдельный файл с локальной базой знаний, они подсвечиваются голубым цветом. Выбрав одиночным щелчком мыши такой объект, можно нажать кнопку **Topic Tree**. После этого текущий контекст запоминается, загружается база знаний для новой темы и вызывается из памяти запомненный контекст, если данная тема уже обсуждалась.

Так, нажав кнопку **Topic Tree**, мы получаем граф переходов от темы к теме, которые происходили в ходе диалога. Таким образом, можно вернуться к любой ранее обсуждавшейся теме, «вспомнив» ее контекст.

Замечание: Строго говоря, граф переходов будет древовидным только в случае соответствующего развития диалога. Если перескакивать с темы на тему беспорядочным образом, граф тем будет иметь сетевую структуру.

7 Сообщения программы

Программа выдает сообщения, относящиеся к диалогу, в поле вывода, а все прочие сообщения – в окне **Messages**.

7.1. Сообщения в поле вывода

Сообщение	Причина	Действия пользователя
No more facts in the current branch.	Была нажата кнопка DEEP, но последний объект является листом	Попытаться пойти по другой ветви
No more facts about the cur rent subject.	Была нажата кнопка WIDE, от последнего субъекта больше нет ветвлений	Попытаться пойти по другой ветви или углубиться в текущей
The fact about <Topic> is f orgotten forever.	Нажата кнопка FORGET FACT. Факт, включающий в себя <Topic>, исключен из контекста и из базы знаний. Если после этого нажать SAVE, то и из файла данной	Любые

	темы	
The fact about <Topic> can not be forgotten.	Нажата кнопка FORGET FACT, а фактов, включающих в себя <Topic>, нет в контексте	Возможная ошибка в программе. Обратиться к автору
Double click the mouse to c hoose an object, then push Forget Fact button.	Нажата кнопка FORGET FACT, а субъект или объект не выбран	Выбрать субъект или объект двойным щелчком мыши
Click the mouse to choose a n object, then push Change Topic button	Нажата кнопка GOTO TOPIC, а субъект или объект не выбран	Выбрать субъект или объект двойным щелчком мыши
The topic <Topic> already exists in the tree	Нажата кнопка GOTO TOPIC для перехода к теме <Topic>, которая уже обсуждалась	Любые
No external knowledge bas e about the topic <Topic>	Нажата кнопка GOTO TOPIC для перехода к теме <Topic>. Внешнего файла для данной темы не создано	Проверить, действительно ли внешний файл был создан, но не подключился
The context is cleared. You may start from scratch.	Нажата кнопка CLEAR ALL. Контекст очищен. База знаний загружена заново.	Любые
Cannot re-initialize the knowledge base!	Нажата кнопка CLEAR ALL. Контекст очищен. База знаний не может быть загружена заново.	Возможная ошибка в программе. Обратиться к автору

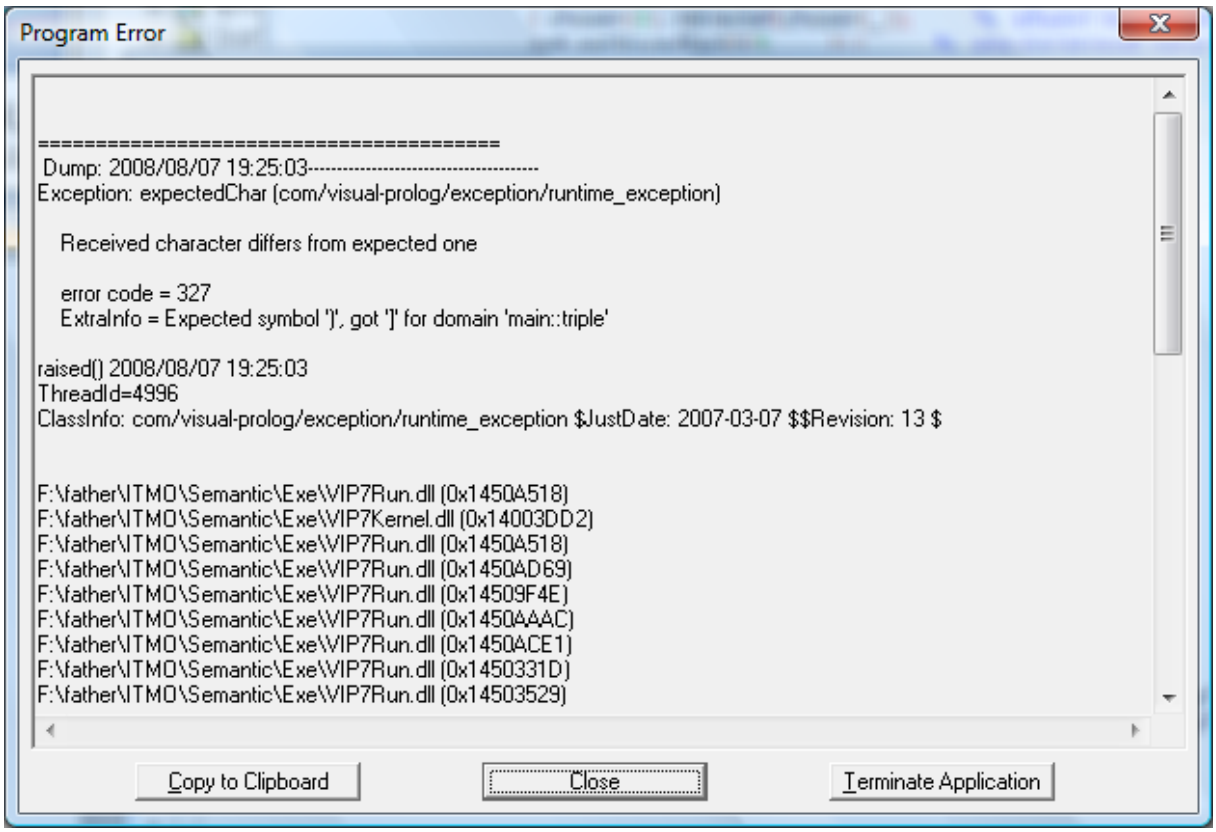
7.2. Сообщения в окне Messages

Сообщения в окне Messages имеют буквенную и цифровую идентификацию. Буква обозначает характер сообщения: i – информационное, W – предупреждение о возможной ошибке, E – сообщение об ошибке.

ID	Текст	Причина	Действия
001i	The file <path\name.txt > is loading….	Файл базы знаний загружается	Не требуется
002 E	Error loading a knowle dge base!	Невозможно загрузить файл базы знаний	Проверить наличие и имя файла, указан-ного в сообще-нии 001i
003i	The file <path\name.txt > is loading….	Файл онтологии загружается.	Не требуется
004 E	Error initialization a knowledge base!	Ошибка загрузки онтологий	Проверить файлы, указанные в записях **onto**
005 E	Error indexing a new fact!	Ошибка индексации факта. Вероятнее всего, ошибка в программе	Обратиться к разработчику
009i	The fact SPO <Term1> <Term2> <Term3> replenished the knowledge base <Subject> <Predicate> <Object>	Предложение пользователя пополнило контекст фактом <Subject> <Predicate> <Object>	Не требуется
025i	A new fact established <Subject> <Predicate> <Object>	К текущему контексту применено правило, записанное в соответствующей строке типа r и получен новый факт <Subject> <Predicate> <Object>	Не требуется
030i	The fact <Subject> <Predicate> <Object> replenished the context	Факт <Subject> <Predicate> <Object> пополнил контекст	Не требуется
040 W	Pre-selection the fact <Subject> <Predicate> <Object> failed	Для данного триплета нет фактов в базе знаний	Не требуется
041i	чч.мм.сс.Forward chai ning started	Начата обработка всех правил	Не требуется
042i	чч.мм.сс.Forward chai ning ended. Rules foun d %, tried %, applied %	Закончена обработка правил. Найдено, использовано, использовано успешно.	Не требуется

071 E	Cannot draw fact <Subject> <Predicate> <Object>	Невозможно вывести на экран факт <Subject> <Predicate> <Object>	Обратиться к автору
072 E	Cannot draw <Object> at <X1,Y1 >	Невозможно вывести на экран объект <Object>	Обратиться к автору
080i	The last fact <Subject> <Predicate> <Object> discarded.	Факт <Subject> <Predicate> <Object> удален из контекста	Не требуется
081i	Current topic is <Topic >.	Основная тема теперь <Topic>.	Не требуется
083i	Selected fact by a click is <Subject> <Predicate> <Object>	Выбран факт Subject> <Predicate> <Object>	Не требуется
091 W	No external knowledge base about the topic <Topic>.	Нет внешней базы данных для темы <Topic>. Если файл должен быть, то возможна ошибка имени файла	Проверить имя файла, если он есть
092i	No context for the topic <Topic>.	Для темы <Topic> нет сохраненного контекста. Если тема уже обсуждалась, то возможна ошибка программы	Обратиться к автору
093i	The topic <Topic> has not been discussed yet.	Тема <Topic> еще не обсуждалась	Не требуется
094i	Now we talk about <Topic>.	Основная тема теперь <Topic>.	Не требуется
095i	The topic "", Topic, "" has been restored.	Контекст основная тема <Topic> восстановлен.	Не требуется
096i	We talk about "", Topic, ""again.	Выполнен повторный переход к ранее сохраненной теме <Topic>.	Не требуется
097i	External knowledge base <base> about the topic <topic> is loaded.	Внешняя база знаний для темы <Topic> загружена	Не требуется

В случае ошибок в синтаксисе файлов локальных или глобальной базы данных выдаются сообщения следующего вида:



К сожалению, данное сообщение не содержит указания на конкретную строку, в которой обнаружена ошибка. Подсказкой может быть строка «Expected symbol ‘)’, got ‘]’”, из которой следует, что где-то стоит квадратная скобка вместо круглой. Для ускорения поиска подобных ошибок для редактирования файлов баз знаний и онтологий целесообразно пользоваться редакторами в составе компиляторов языка Пролог, например, SWI-Prolog или Visual Prolog.

СПбГУ ИТМО стал победителем конкурса инновационных образовательных программ вузов России на 2007–2008 годы и успешно реализовал инновационную образовательную программу «Инновационная система подготовки специалистов нового поколения в области информационных и оптических технологий», что позволило выйти на качественно новый уровень подготовки выпускников и удовлетворять возрастающий спрос на специалистов в информационной, оптической и других высокотехнологичных отраслях науки. Реализация этой программы создала основу формирования программы дальнейшего развития вуза до 2015 года, включая внедрение современной модели образования.

КАФЕДРА ВЫЧИСЛИТЕЛЬНОЙ ТЕХНИКИ

Кафедра вычислительной техники СПбГУ ИТМО создана в 1937 году и является одной из старейших и авторитетнейших научно-педагогических школ России. Заведующими кафедрой в разное время были выдающиеся деятели науки и техники М.Ф. Маликов, С.А. Изенбек, С.А. Майоров, Г.И. Новиков. Многие поколения студентов и инженеров в Советском Союзе и за его пределами изучали вычислительную технику по учебникам «Структура ЭВМ» и «Принципы организации цифровых машин» С.А. Майорова и Г.И. Новикова, «Введение в МикроЭВМ» В.В. Кириллова и А.А. Приблуды, «Основы теории вычислительных систем» С.А. Майорова и Г.И. Новикова, А.А. Приблуды, Б.Д.Тимченко, Э.И. Махарева и др.

Основные направления учебной и научной деятельности кафедры в настоящее время включают в себя встроенные управляющие и вычислительные системы на базе микропроцессорной техники, информационные системы и базы данных, сети и телекоммуникации, моделирование вычислительных систем и процессов, обработка сигналов.

Выпускники кафедры успешно работают не только в разных регионах России, но и во многих странах мира: Австралии, США, Канаде, Германии, Индии,

Китае, Монголи, Польше, Болгарии, Кубе, Израиле, Камеруне, Нигерии, Иордании и др.

Игорь Александрович Бессмертный

ИСКУССТВЕННЫЙ ИНТЕЛЛЕКТ

Учебное пособие

В авторской редакции
Дизайн И.А. Бессмертный
Верстка И.А. Бессмертный
Редакционно-издательский отдел Санкт-Петербургского государственного университета информационных технологий, механики и оптики
Зав. РИО Н.Ф. Гусарова
Лицензия ИД № 00408 от 05.11.99
Подписано к печати
Заказ №
Тираж 100 экз.
Отпечатано на ризографе

Редакционно-издательский отдел
Санкт-Петербургского государственного
университета информационных технологий,
механики и оптики
197101, Санкт-Петербург, Кронверкский пр., 49