Санкт-Петербургский государственный университет информационных технологий, механики и оптики

Учебно-методическое пособие по дисциплине «Системы представления знаний»

Новиков Ф.А.,
канд. физ.-мат. наук, доцент кафедры «Технологии программирования»

Санкт-Петербург
2007

Оглавление

Аббревиатуры

СПЗ –	Системы Представления Знаний
ИИ –	Искусственный Интеллект
ПСИИ –	Прикладная Система с элементами ИИ
СП –	Система Продукций
ИП –	Исчисление Предикатов
АДТ –	Автоматическое Доказательство Теорем
МР –	Метод Резолюций
АСП –	Автоматический Синтез Программ
СПТ –	Система Подстановки Термов

Введение

Современное состояние в области применения компьютеров характеризуется возрастанием значения методов искусственного интеллекта (ИИ) в программном обеспечении. В настоящее время происходит непрерывный процесс «интеллектуализации компьютеров». Системы представления знаний, таким образом, являются наиболее актуальным направлением программирования.

Инженеры, подготовленные по специальности «прикладная математика», должны владеть основными, уже устоявшимися методами поиска решения переборных задач, разработанными в исследованиях по искусственному интеллекту, и основными связанными с этими методами формализмами, используемыми для предоставления знаний в компьютерах.

Направление искусственного интеллекта в информатике с самого начала вызвало большой, можно даже сказать, нездоровый интерес. В последнее время результаты, полученные в этом направлении, переросли рамки чисто теоретических, экспериментальных разработок и начали применяться в практических приложениях. Поэтому курсы «Системы представления знаний», «Прикладной искусственный интеллект» и т.п. стали практически обязательными в учебных планах подготовки специалистов по информатике.

Данная дисциплина логически завершает цикл курсов в области прикладного программного обеспечения. В ней активно используется материал по математической логике, базам данных и языкам программирования.

Важная особенность данного учебника заключается в том, что изложение ведется большей частью на неформальном уровне, с большим количеством примеров. Основное внимание уделяется объяснению идей, а не формальным определениям. Все излагаемые алгоритмы описываются с помощью неформального псевдокода. Таким образом, предполагается, что читатель уже обладает достаточно программисткой интуицией, чтобы понимать программы без формального определения синтаксиса, и владеет языком исчисления предикатов первого порядка достаточно свободно, чтобы понимать примеры.

Целью данного курса является достижение обучаемыми следующих результатов.

1. Знание основных методов поиска решения переборных задач; знание сравнительных достоинств и недостатков этих методов; знание основных принципов наиболее известных способов представления знаний: правил продукции, формул исчисления предикатов, семантических сетей.
2. Умение выбрать подходящий способ представления знаний для конкретной задачи в простых случаях; умение оценить адекватность использования для конкретной задачи того или иного метода поиска решения.
3. Свободное владение основными понятиями из области искусственного интеллекта и представления знаний. Умение изучить по литературе другие методы искусственного интеллекта. Умение оценить и воспользоваться конкретной прикладной системой искусственного интеллекта.

Методы ИИ сейчас очень быстро развиваются и меняются, и невозможно дать заранее готовые рецепты на все случаи жизни программиста. Вместо этого дается общее представление об основных принципах ИИ и обзор наиболее интересных в настоящее время приложений.

Тема 1. Прикладные системы с искусственным интеллектом

Данный учебник называется «Системы представления знаний». Другими словами, он должен давать ответ на вопрос: как знания и умения человека выразить в виде программы для компьютера?

Считается, что многие виды умственной деятельности человека, например: программирование, занятие математикой, чтение лекций требуют «интеллекта». Мы видим, что и для человека понятие «интеллект» определяется косвенно, через предметную область по некоторым результатам деятельности, а не напрямую указывает, где конкретно в человеке находится интеллект. Другими словами, определения ничего не определяют, но могут иногда стимулировать верные ассоциации.

Программа, умеющая решать задачи в предметной области, которую традиция относит к интеллектуальным, называется *прикладной системой с элементами ИИ* (*ПСИИ*).

Другое определение: ПСИИ – это программа, алгоритм которой нельзя найти в вузовском учебнике. В каждом отдельном случае это ноу-хау создателей такого ПО, но в любом случае любая новая идея дает существенное продвижение в этой области.

Для мотивации дальнейшего изложения рассмотрим некоторые из предметных областей, типичных для ИИ.

1.1. Обзор приложений ИИ

Некоторыми типичными приложениями, использующие элементы ИИ, являются:

- понимание естественного языка;
- машинный перевод;
- интеллектуальные базы данных;
- экспертные системы;
- автоматическое доказательство теорем;
- планирование действий робота;
- распознавание образов;
- интеллектуальные игры.

1.1.1. Понимание естественного языка

Процесс понимания естественного языка (ЕЯ) – это процесс передачи данных от одного человека к другому. Несмотря на всю свою кажущуюся простоту, это процесс чрезвычайно сложен. Он подразумевает наличие общего контекста знаний у обоих партнеров и согласованных механизма понимания, используя мощные вычислительные ресурсы, связанные с интеллектом разговаривающих (рис. 1).

Общий контекст понимания – это совокупность согласованных механизмов (алгоритмов) представления (вывода) данных одним и последующей интерпретацией другим партнером.

Мудрецу достаточно сказать одно слово, для того, чтобы другой мудрец его понял. Иными словами, для мудрецов общий объем передаваемых данных может быть значительно меньше, чем у среднестатистических людей, за счет более мощного механизма интерпретации данных.
Существует два принципиально разных способа общения человека с компьютером. Первый из них, когда человек способен понимать компьютер. Это не заслуга компьютера, а способность человека, которого все окружающие называют программистом. Второй, когда компьютер способен понимать человека. Такой компьютер обязательно должен обладать элементами ИИ. И такие программы были созданы уже сравнительно давно.

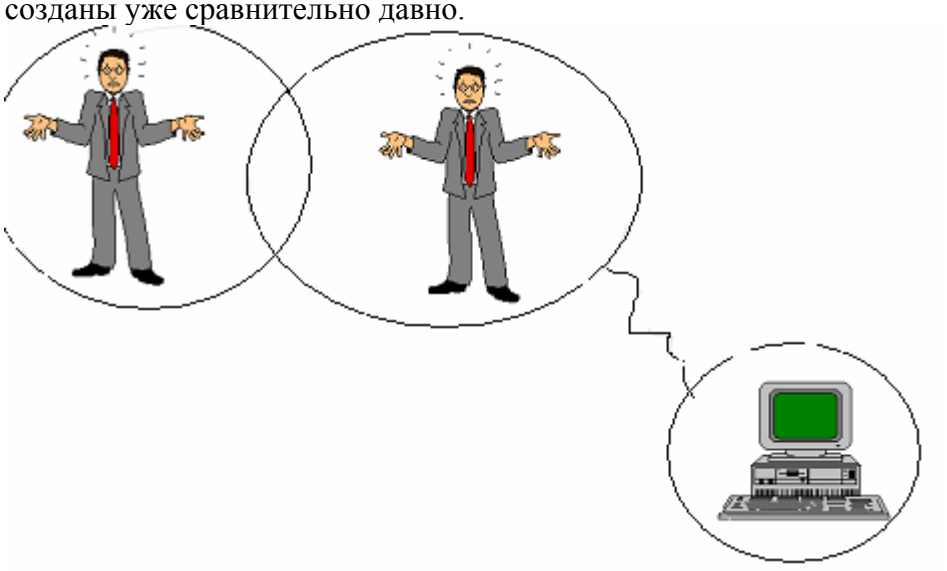


Рис. 1. Общий контекст понимания

Одной из первых ПСИИ, понимающих естественный язык, явилась созданная в 1966 году Дж. Вейценбаумом программа ELIZA.

Джозеф Вейценбаум (*Joseph Weizenbaum*), *род.* 1923 — *известный деятель в области искусственного интеллекта, заслуженный профессор МТИ* (*Массачусетский технологический институт*) *в области computer science.*
Основатель организации «Компьютерные профессионалы за социальную ответственность».

Позднее Виноград написал книгу «Программа, понимающая естественный язык», описывающую ELIZA. Структура ELIZA была чрезвычайно проста и включала в себя всего несколько десятков эвристических правил, предназначенных для распознавания грамматических конструкций. В результате эта программа могла вести вполне правдоподобный диалог, зачастую, при этом, отвечая вопросом на вопрос. **Пример** диалога с ELIZA:

```
- ELIZA, ты компьютерная программа.
- Почему Вы думаете, что я компьютерная программа?
```

```
- Потому что тебя составил Вейнценбаум.
- Расскажите мне о Вейнценбауме.
```

Несмотря на всю свою простоту, эта программа находит применение в психиатрии. Разговор психиатра с невротиком продолжается не более 20 минут, после чего врачу просто необходим отдых. Программа ELIZA способна общаться с больным часами, при этом наблюдаются явные улучшения в психическом состоянии невротика.

1.1.2. Машинный перевод

Важный класс примеров, связанных с пониманием естественного языка, образуют системы автоматического перевода текстов с одного естественного языка на другой.

Важно, что, как и в других случаях ИИ, в классе систем машинного перевода имеется непрерывный спектр систем все возрастающей сложности:

- электронный словарь;
- подстрочник технических текстов;
- художественный перевод поэзии;
- синхронный перевод речи.

Для каждой новой области применения ИИ наблюдается одна и та же характерная картина, которую мы проиллюстрируем на примере машинного перевода. После появления новой идеи к ней проявляется повышенный интерес, и возникают завышенные ожидания, пользователи ждут чуда. Но чуда не происходит – методы ИИ трудоемки, сложны и их развитие требует немалого естественного интеллекта. Наступает разочарование, новое направление уходит из сферы ажиотажного интереса и иногда подвергается даже необоснованной обструкции. Если идея была плодотворна, то, несмотря на падения интереса в средствах массовой информации, подспудная кропотливая работа в академических кругах продолжается и через некоторое время (не сразу!) начинает давать практические результаты. Интерес вновь появляется, но уже не праздный, а деловой и обоснованный.

История развития машинного перевода насчитывает более полувека и начинается в 1947 г с появлением электронного словаря (рис. 2)

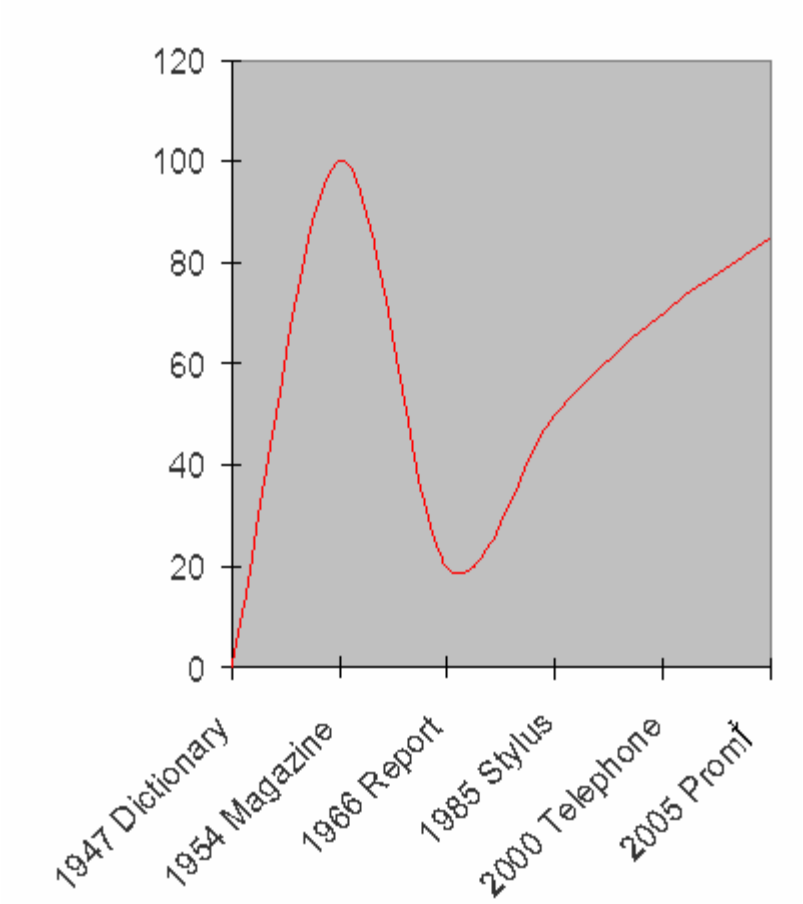


Рис. 2. История популярности машинного перевода

Первый максимум в истории развития машинного перевода проявляется в 1954 – дата создания международного специализированного журнала. В этот период сотни исследовательских групп по всему миру занимались данной проблемой. В 1966 был отмечен мощный спад, связанный с отчетом Министерства обороны перед конгрессом США, в котором было сказано, что машинный перевод практически неосуществим, а потому Министерство обороны прекращает финансирование такого рода проектов. Тема перестала быть модной и работу продолжили только квалифицированные энтузиасты и профессионалы. Постепенно, к настоящему времени разработки в этой области выходят на прежний уровень ожиданий – сейчас машинный перевод технических и не слишком сложных деловых текстов достиг вполне удовлетворительного уровня качества и ведутся интенсивные работы по синхронному переводу речи (например, в Интернет телефонии).

Впрочем, мечты, с которыми род людской взялся полвека назад за задачу машинного перевода, в значительной мере остаются мечтами: высококачественный перевод текстов широкой тематики по-прежнему недостижим. Однако несомненным является ускорение работы переводчика при использовании систем машинного перевода: по оценкам конца 1980-х, до пяти раз.

В настоящее время существует множество коммерческих проектов машинного перевода. Одним из пионеров в области машинного перевода была компания Systran. В России большой вклад в развитие машинного перевода внесла группа

под руководством проф. Р. Г. Пиотровского (Российский государственный педагогический университет им. Герцена, Санкт-Петербург).
Сейчас заслуженное признание получил автоматический переводчик, разработанный компанией ПРОМТ.

1.1.3. Интеллектуальные базы данных

Интеллектуальные базы данных (Intelligent Databases) предоставляют эффективные способы хранения, представления и извлечения огромного числа фактов. Более того, такие базы позволяют отвечать на вопросы, требующие дедуктивного рассуждения.
Идея состоит в следующем. Допустим, что кроме обычных реляционных таблиц (см. табл. 1) в базе данных хранятся и некоторые правила, например, такое

If `x.Должность = "Зав.*" & x.Работа = y.Работа`
Then `x начальник y`

Приведенный пример позволяет выяснить отношения на уровне начальник-подчиненный. В данном случае, очевидно, что Клавдиев является начальником не только у Новикова, но и у любого другого сотрудника кафедры «Прикладная математика».

ФИО	**Должность**	**Подразделение**
Новиков	Доцент	Кафедра «Прикладная математика»
Клавдиев	Заведующий	Кафедра «Прикладная математика»

Таблица 1. Таблица базы данных.

1.1.4. Экспертные системы

Методы интеллектуальных баз данных применяются также при разработке автоматических консультирующих систем. По своей сути, эти системы способны предоставить компетентный ответ пользователю на конкретный вопрос.
Такие экспертные системы (Expert Systems) применяются, например, в медицине (диагностика заболеваний) и в геологии (оценка месторождений).
Одна из самых главных проблем в этой области заключается в том, как следует представлять и использовать знания экспертов, которые по своей сути являются в достаточной степени разнородными и противоречивыми данными.

Экспертные системы – это набор фатов и эмпирические правила манипулирования данными фактами. Считается, что экспертные системы должны объяснять ход рассуждения.

Структура экспертных систем состоит из пяти модулей. Совокупность трех модулей –Правила, Модель ситуации и Вывод – по сути, являются интеллектуальной базой данных. Модули, позволяющие извлекать знания и объяснять выводы, превращают обычную базу данных в экспертную систему.

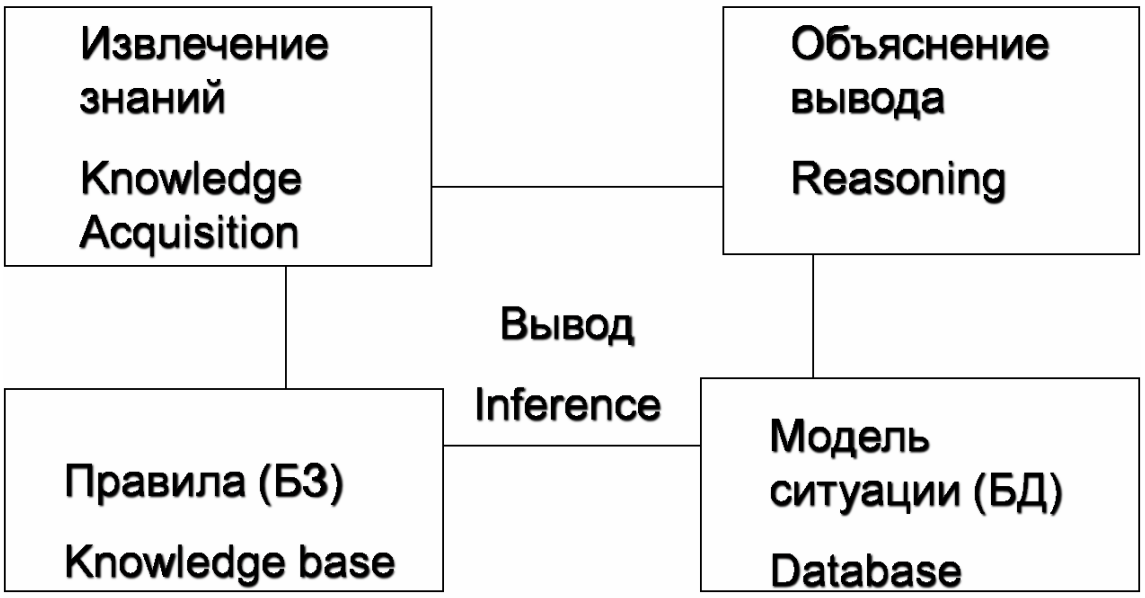


Рис. 3. Архитектура экспертной системы

Сейчас обычно база знаний для ЭС – это набор правил. Большинство выводов, сделанных ЭС, имеют вероятностный характер. Достаточно часто объяснения имеют следующий вид:

```
если …, то … с вероятностью …
```

1.1.5. Автоматическое доказательство теорем

Доказательство теорем безусловно предполагает затрату интеллектуальных усилий, требующий не только дедуктивных навыков, но и в значительной степени интуитивных предпосылок (например, какие леммы необходимы для той или иной теоремы). Другими словами, доказательство теорем подразумевает, в том числе, поиск уже доказанных теорем и решенных подзадач, которые могут оказаться полезны для достижения конечной цели. К настоящему времени разработано несколько программ с элементами ИИ, то есть программ автоматического доказательства теорем (Automatic Theorem Proving) способных работать в этом направлении.

При первом взгляде на предмет бросаются в глаза три обстоятельства, известные из курса математической логики.

- Универсальность языка исчисления предикатов первого порядка (то есть любое утверждение может быть сформулировано как формула прикладного исчисления предикатов).
- Алгоритм Тарского перечисления правильно построенных формул (то есть все правильно построенные формулы могут быть перечислены).
- Полнота метода резолюций (то есть вывод доказуемой формулы может быть найден).

Таким образом, автоматическое доказательство теорем возможно, и его нетрудно запрограммировать, но такой прямолинейный подход оказывается безнадежно неэффективным. Сложные, интересные теоремы автоматически доказать не удается, потому что пространство перебора столь велико, что даже самые

быстродействующие компьютеры не в состоянии ничего сделать за разумное время.

Однако существуют примеры, когда в очень узком классе математических теорий, применяя очень специфические методы, удавалось автоматически находить и доказывать нетривиальные содержательные теоремы, которые были ранее неизвестны.

1.1.6. Автоматическое управление роботом

Управление роботом (Robot Planning) – задача чрезвычайно сложная. Казалось бы, нет ничего проще, чем передвигаться на двух ногах. Развитие любого человеческого существа начинается со способности видеть и ходить – тех навыков, которые вроде бы и не требуют значительных интеллектуальных усилий (рис. 4). На первый взгляд может показаться, что говорить, рассуждать и программировать значительно сложнее. Однако если взглянуть на «развитие» роботов и робототехники, можно легко убедиться в обратном. Роботы сначала научились программировать и рассуждать, потом говорить.

Успешное хождение двуного робота – достаточно сложная задача даже для современного развития науки и техники. Дело в том, что когда мы ходим, центр тяжести нашего тела почти никогда не находится над площадью опоры. С точки зрения механики, мы не ходим, а непрерывно падаем, но не совсем, а удачно подставляем ногу в нужное место и продолжаем движение. Лобовой подход — составить кинематическую модель с нужным количеством степеней свободы и на каждом шаге быстро решать соответствующую систему дифференциальных уравнений не работает. Даже самый быстродействующий современный компьютер не успевает проводить нужные вычисления в реальном времени. Для реализации двуного хождения роботов применяются совсем другие приемы. Что касается бега, то двуногий бегающий робот — почти нереальная задача для современного уровня развития науки и техники.

Утрируя, можно сказать, что обезьяна превратилась в человека не потому, что стала ходить на двух ногах, а скорее человек стал достаточно умен, чтобы на двух ногах начать передвигаться.

До сих пор не создано адекватного аппарата, позволяющего роботу «видеть» окружающий мир и реагировать на «визуальные» изменения в нем. Известно, что на третьей неделе развития ребенок обладает цветным стереоскопическим зрением. Важно подчеркнуть, что эта способность объясняется не физиологическими, а интеллектуальными причинами. В этом нетрудно убедиться: прикройте один глаз — способность видеть не плоский, а трехмерный мир и определять расстояния до предметов сохранится. С другой стороны, биологи установили, что если поместить человека в совершенно непривычную искусственную среду (в которой ненормально устроено освещение, привычные предметы имеют непривычные размеры, процессы имеют ненормальный темп и т.д.), то способность к адекватному восприятию трехмерной действительности человеком утрачивается. Таким образом, цветное стереоскопическое зрение – это способность мозга, а не глаза. Мы видим не то, что есть на самом деле, а то, что ожидаем увидеть, ту модель, которую очень быстро, за десятые доли секунды, строит мозг по информации, полученной из глаза. Проблема технического зрения заключается не в разрешающей способности камер и не в их количестве, а в недостатке алгоритмов,

адекватно восстанавливающих трехмерную модель по совокупности плоских изображений, полученных от камер.

Рис. 4. Сравнение возможностей робота и человека

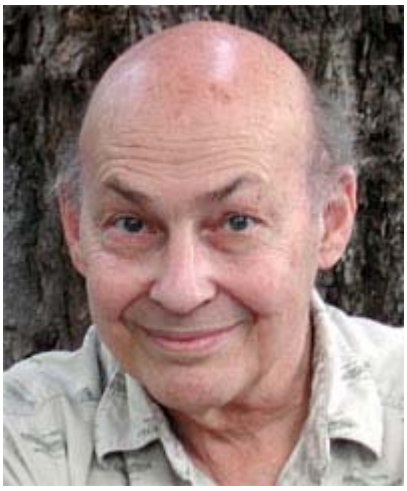
1.1.7. Распознавание образов

Распознавание образов (Pattern recognition) – классическая, едва ли не самая первая область применения методов ИИ. Можно упомянуть такие исследования, как персептрон Розенблатта – одна из первых искусственных сетей, способных к перцепции (восприятию) и формированию реакции на воспринятый стимул. Персептрон рассматривался его автором не как конкретное техническое вычислительное устройство, а как модель работы мозга. Нужно заметить, что после нескольких десятилетий исследований современные работы по искусственным нейронным сетям редко преследуют такую цель.

Розенблатт Фрэнк, (*Rosenblatt Frank*) *род.* 1928 — *американский психолог. До* 1955 *работал в национальной корпорации здравоохранения и в исследовательском центре социальных наук, затем в Корнеллской аэронавтической лаборатории* (*с* 1959 *директор исследовательской программы по проблеме распознавания образов*). *В* 1957 *предложил и реализовал принцип персептрона.*

На примере распознавания образов мы опять наблюдаем характерную кривую развития направления ИИ. После появления пионерской работы Розенблатта

наблюдался резкий подъем интереса к этой тематике, затем последовал холодный душ в виде теоремы Минского о принципиальной ограниченности персептронов, казалось, что перспективы утрачены, но к настоящему времени первоначальная идея трансформировалась в нейронные сети и нейрокомпьютеры, которые более чем успешны.



Ма́рвин Ли Ми́нский, (*Marvin Lee Minsky*) *род.* 1927 — *профессор Массачусетского технологического института с* 1958. *Лауреат премии Тьюринга* 1969 *года, премии Японии* 1990 *года, премии «за научные достижения» Международной конференции по искусственному интеллекту* 1991 *года, медали института Бенджамина Франклина* 2001 *года.*
Обладатель патентов на головной графический дисплей (1963) *и конфокальный сканирующий микроскоп* (1961*, предшественник современных широко распространённых конфокальных лазерных сканирующих микроскопов*). *Вместе с Сеймуром Папертом создал первую «черепашку» на языке Logo. В* 1951 *сконструировал первую обучающуюся машину со случайно связанной нейросетью — SNARC.*

Один из первых серьезных успехов в распознавании образов (или, другими словами, прообраза машинного зрения) связан с созданием в 1982 г фирмой Logica системы распознавания отпечатков пальцев, которая находит широкое применение не только в криминалистике, но и в разнообразных системах защиты, например в ноутбуках.
Другими примерами применение ПСИИ является распознавание речи, распознавание сканированного текста (OCR) или мониторинг среды по аэрофотоснимкам.
При этом необходимо хорошо понимать, что синтез объекта всегда значительно проще, чем анализ того же самого объекта. Наглядный пример – рукописный текст. Его очень легко написать, но прочитать бывает иногда неимоверно трудно.

1.1.8. Интеллектуальные игры

Одно из самых известных и популярных применений ПСИИ – это самые разнообразные интеллектуальные игры (Intelligent games). Почти наверняка каждый сталкивался с искусственным интеллектом, трудящимся именно в этой области. Шашки, шахматы, нарды встречаются и в мобильном телефоне, и в качестве более серьезных программ для достаточно мощных компьютеров. Достаточно сказать, что современные компьютерные шахматные программы играют на одном уровне с ведущими гроссмейстерами мира, и, иногда, даже побеждают. При этом компьютерные шахматы – это не только развлечение, но и способ обучения новых шахматистов, создание шахматных баз данных, анализ

окончаний, изменение правил.[1] Но самое главное, в любом случае, это один из наиболее интересных способов представления знаний.

Задача представления знаний в интеллектуальных играх является очень сложной. Дело в том, что некоторые интеллектуальные игры чрезвычайно сложны с количественной точки зрения.

Например, шашки имеют порядка 10^{40} позиций, а шахматы – 10^{120} позиций. Это огромные числа, большие, чем число атомов во Вселенной и чем число наносекунд, оставшихся до конца света.

Уже несколько десятилетий проводятся регулярные чемпионаты мира среди шахматных программ. Отрадно отметить, что на первых чемпионатах побеждала отечественная программа «Каисса», разработанная под руководством Адельсона-Вельского и Донского. Сейчас первенство утрачено, но … . Огромный вклад в алгоритмическое понимание шахмат внес М. М. Ботвинник.

Георгий Максимович Адельсон-Вельский, *род.* 1922 *— советский математик. Вместе с Е. М. Ландисом в* 1962 *изобрёл структуру данных, получившую название АВЛ-дерево. С* 1957 *занимался проблемами искусственного интеллекта, в* 1965 *руководил разработкой компьютерной шахматной программы в Институте теоретической и экспериментальной физики, которая победила американскую программу Kotok-McCarthy на первом шахматном матче между компьютерными программами; впоследствии на её основе была создана программа «Каисса», в* 1974 *ставшая первым компьютерным чемпионом мира по шахматам на чемпионате в Стокгольме.*

Шахматное программирование достигло очень высокого уровня. Если провести турнир между обычной студенческой группой и обычной шахматной программой, например, ChessMaster, то окажется, что в среднем группа играет хуже, чем Chess Master. О чем это говорит?

1.2. Место представления знаний в искусственном интеллекте

В любой интеллектуальной области невозможно создать универсальный алгоритм, подходящий для решения любой задачи. Метод решения – это всегда метод проб и ошибок, поиск новых подходов, новых путей, в конце концов – банальный перебор возможных решений.

[1] Любопытный пример. Ранее в шахматах действовало правило, что партия заканчивается вничью, если за пятьдесят ходов не было взятий или превращений. В результате компьютерного анализа было найдено результативное окончание, которое требовало 53 хода. В результате шахматные правила были изменены.

1.2.1. Итеративный характер решения задач

Общая схема решения проблемы с помощью компьютера изображена на рис. 5. Стрелочки сверху вниз – детерминированный процесс, стрелочки снизу вверх – недетерминированный. Решение проблемы всегда носит, в целом, недетерминированный характер. Это метод проб и ошибок, метод поиска, в результате которого часто приходится возвращаться на предыдущие шаги.

Если обратные связь в таком процессе (стрелочки снизу вверх) реализуется через человека (а это происходит достаточно часто), то такой процесс не может называться обработкой знаний. Для того чтобы можно было бы говорить именно о компьютерной обработке знаний, каждая петля обратной связи должна замыкаться, соответственно, через компьютер. И тогда данные, с которыми работает такая программа, вправе называться знаниями.

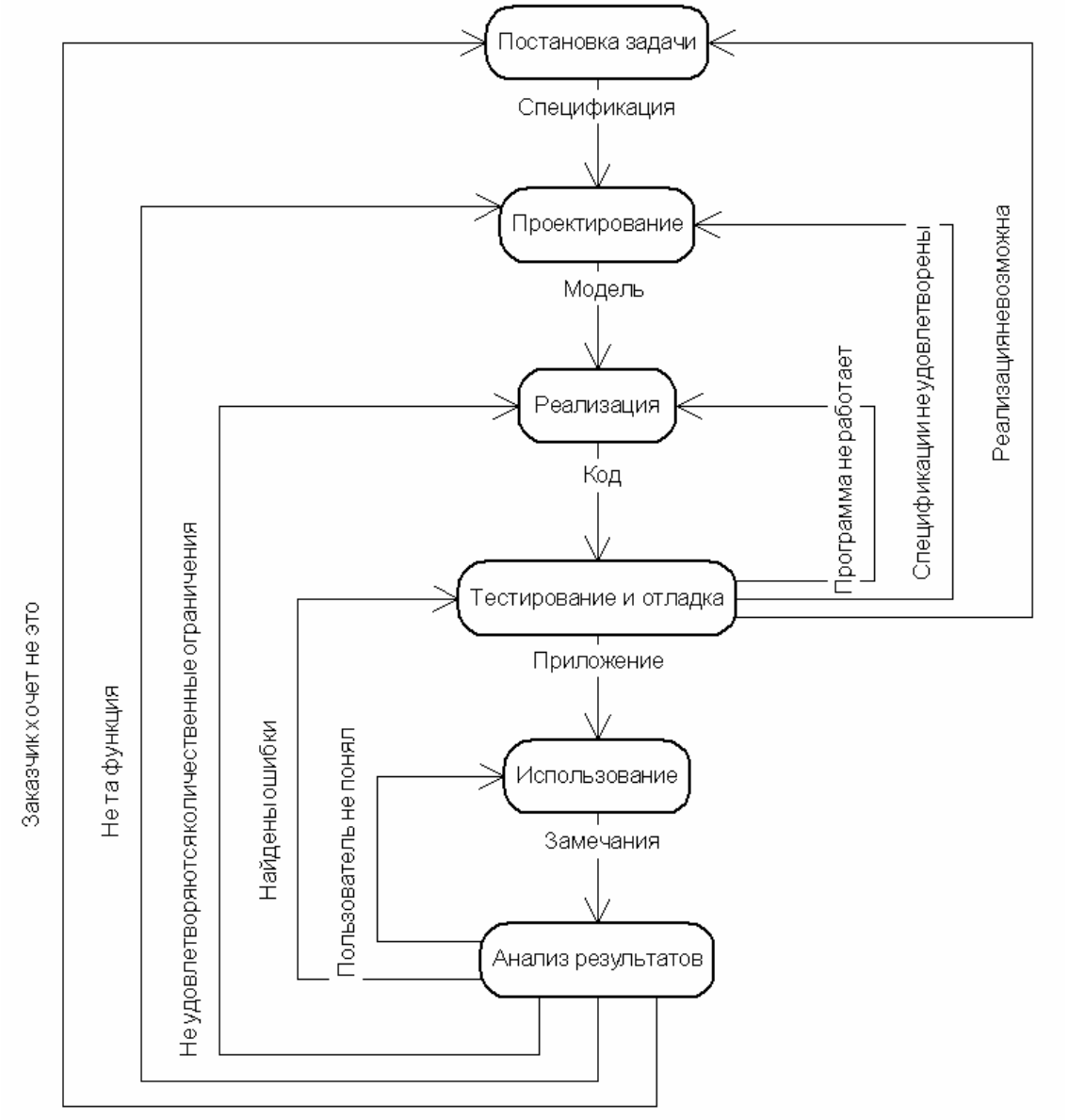


Рис. 5. Схема процесса решения интеллектуальной задачи

При традиционные процедурном программировании (сверху вниз), решение целиком зависит от задачи. Каждая задача имеет свое решение, и решение одной не может быть непосредственно, без дополнительной работы, использовано для решения другой задачи. В ИИ (непроцедурное программирование) один и тот же метод поиска может применяться к целому набору проблем.
На самом деле, четкой границы между этими двумя методами поиска нет. Любая процедура обязательно содержит знания, а любые знания представляются процедурой.

1.2.2. Знание и незнание

Явно выраженную границу между знанием и незнанием указать невозможно.
На самом деле точной границы для представления знаний в компьютере нет – всякая программа представляет какие-то знания. Точнее говоря, ничто не есть знания без нашей собственной интерпретации. Рассмотрим это на модельном примере.
Допустим, мы хотим, чтобы компьютер отсортировал набор чисел в порядке их возрастания. Пусть нам заранее известно, что этих чисел три и они равны 3,5 и 7. Можно научить компьютер решать данную задачу, написав всего один оператор:

```
Write (3, 5, 7).
```

Таким образом, мы представили в компьютер знания, достаточные для решения это отдельной задачи.
Усложним задачу. Пусть чисел по-прежнему три, но при этом значение их произвольное. Составим процедуру, способную отсортировать эти числа по возрастанию:

```
proc Sort3 (x, y, z)
     if x > y then x :=: y
     if y > z then y :=: z
     if x > y then x :=: y
     write (x, y, z)
end proc
```

Таким образом, мы научили компьютер решать довольно большое множество задач, связанных с сортировкой трех чисел по возрастанию, не слишком удлинив при этом запись. Во всяком случае, данная запись намного меньше, чем все возможные операторы `Write`. Другими словами, механизм процедур в языках программирования – это уже механизм представления знаний. Просто это достаточно «старый» механизм, мы к нему привыкли и не воспринимаем как механизм представления знаний.
Возникает еще один вопрос – откуда взялась процедура `Sort3`?
Заметим, что это процедура состоит из совершенно аналогичных операторов.[2] Проведем рефакторинг и перепишем ее, введя процедуру `Sort2`.

```
proc Sort3 (x, y, z)
     Sort2(x, y)
```

[2] Отсюда видно, что процедура `Sort3` использует идею метода пузырька.

```
      Sort2(y, z)
      Sort2(x, y)
      write (x, y, z)
end proc
proc Sort2(u, v)
      if u > v then u :=: v
end proc
```

Но процедуру `Sort2` нетрудно специфицировать формально:

```
x, y { Sort2(x, y) } x < y
```

Здесь перед заголовком процедуры (который в фигурных скобках) выписано предусловие, а после заголовка – постусловие.

Тогда, используя эту спецификацию, мы можем логически вывести процедуру `Sort3`:

```
{x, y, z}
Sort2(x, y) {x < y, z}
Sort2(y, z) {y < z, x < z}
Sort2(x, y) {x < y < z}.
```

Таким образом, мы немного усложнили механизм транслятора (допустив возможность логического вывода при трансляции) и получили систему автоматического синтеза программ. Другими словами, мы компактно записали не одну процедуру `Sort3`**,** а целый класс процедур.

1.2.3. Алгоритмы поиска решения и представление знаний

Первый этап развития ИИ был связан с разработкой методов решения задач – алгоритмов поиска решения и построения логического вывода. В настоящее время это уже достаточно проработанный и изученный вопрос.

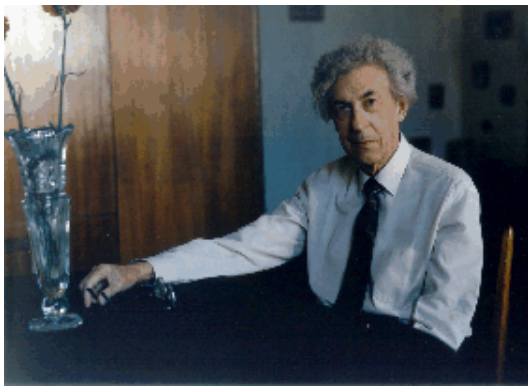
Однако для нетривиальных случаев сильно возрастает трудоемкость поиска решения. Трудоемкость поиска решения – это быстрорастущая функция (в большинстве случаев растущая как экспонента или быстрее), аргументом которой является общий объем базы знаний. Один из способов повышения эффективности поиска решения и уменьшения трудоемкости, является изменение алгоритмов поиска. Однако, на данный момент, серьезного улучшения в этом направлении ожидать трудно. Гораздо более продуктивным методом, позволяющим уменьшить трудоемкость поиска решения, является представление знаний. Если выбор изощренного представления позволяет уменьшить объем базы знаний, то тот же стандартный алгоритм поиска становиться значительно эффективнее в том смысле, что расширяются границы его применимости.

Компактность базы – определяющий фактор эффективности системы и, следовательно, границ применимости ПСИИ.

1.3. Классификации прикладных систем искусственного интеллекта

Сами знания, которые мы собираемся представлять в компьютере, весьма разнообразны. Следуя С.С. Лаврову, знания можно разделить на три принципиально разных вида:

1. *Понятийные, концептуальные знания*. К этой группе относятся понятия и взаимосвязи между ними, например, правила и законы.
2. *Процедурные, алгоритмические знания*. Сюда можно отнести различные умения, технологии, процедуры.
3. *Фактографические знания*. Это самые разнообразные количественные и качественные характеристики конкретных объектов. Например, различные данные.



Лавров Святослав Сергеевич (1923 - 2004) *- российский ученый, член-корреспондент РАН* (1991*; член-корреспондент АН СССР с* 1966). *Труды по механике, автоматическому управлению, вычислительной математике. Ленинская премия* (1957).

Научная биография Святослава Сергеевича Лаврова в определенном смысле уникальна. Совсем в молодом возрасте С. С. Лавров стал основоположником ракетно-космической баллистики в СССР и неоспоримым авторитетом в области динамики управляемого полета и автоматического управления. Появление цифровой вычислительной техники привело к резкому повороту в деятельности С. С. Лаврова и в течение нескольких лет сделало его классиком программирования в СССР.

В зависимости от того, какие виды знаний используются и каким образом, системы ПСИИ можно классифицировать несколькими способами.

1.3.1. Классификация по степени использования различных видов знаний

Если в программной системе используются только фактографические знания, то такую программную систему называют базой данных (БД) и в современных условиях обычно не считают ПСИИ.[3]

[3] Следует заметить, что такие современные технологии баз данных, как извлечение знаний из данных (data mining), скорее следует относить к области ИИ, чем к области систем управления базами данных (СУБД).

Если в программной системе используются главным образом алгоритмические знания, то такую программную систему принято называть пакетом прикладных программ (ППП). Отнесение ППП к ПСИИ обычно явно указывается. Если ППП используется просто как библиотека процедур, то ППП не считают ПСИИ. Если же в ППП используются методы ИИ, например, если программу решения конкретной задачи строит не пользователь ППП, а автоматический планировщик, то такой ППП может считаться ПСИИ.

Если в программной системе в той или иной форме используются концептуальные знания, то такую систему считают ПСИИ.

Заметим, что, как правило, развитые ППП имеют в своем составе или активно используют внешнюю БД, а ПСИИ почти всегда используют как ППП, так и БД (рис. 6).

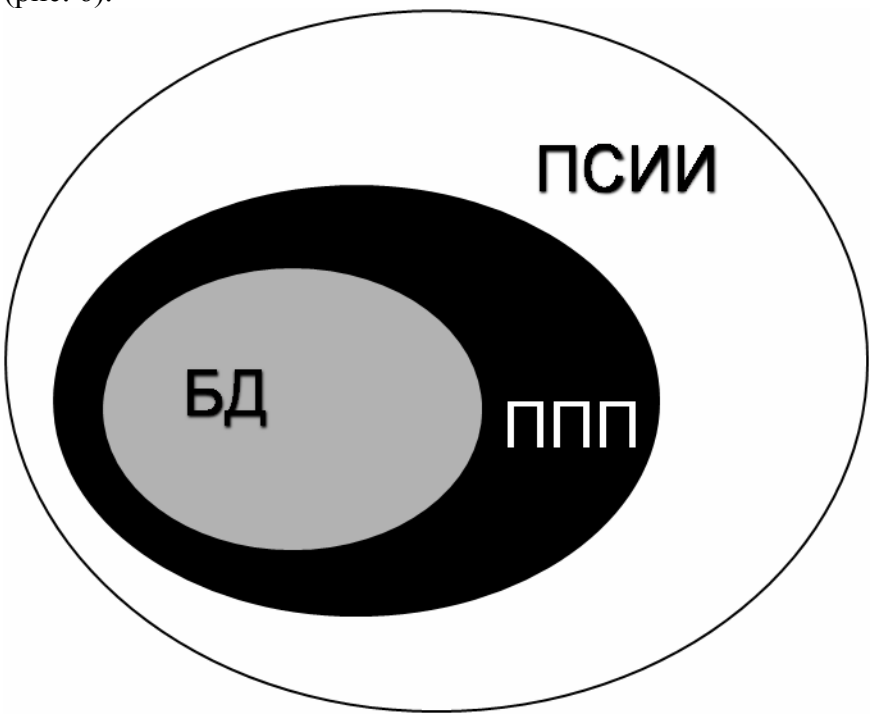


Рис. 6. Классификация ПСИИ по типам знаний

1.3.2. Классификация по форме представления знаний

Когда осознана и прочувствована необходимость совместного использования знаний всех трех типов, возникает потребность в выборе способа их представления в компьютере. Мощные механизмы, которые использует человек для представления знаний – естественные языки, математика, визуальные образы – пригодны для представления всех видов знаний, но непригодны для реализации в компьютере. Точнее говоря, современные компьютеры непригодны для использования этих механизмов – компьютеры слишком примитивны и неинтеллектуальны.

Таким образом, возникает необходимость изобретения таких способов представления знаний, которые, с одной стороны, позволяли бы представлять знания всех видов, и, с другой стороны, были бы реализуемы в современных компьютерах.

Механизмы для представления фактографических и алгоритмических знаний в компьютерах хорошо изучены и устоялись.
Для фактографических знаний – это системы управления базами данных (СУБД), в современных условиях чаще всего реляционные.
Для алгоритмических знаний – это библиотеки процедур, в современных условиях чаще всего объектно-ориентированные.
Таким образом, классификация по форме представления знаний рассматривает, прежде всего, представление концептуальных знаний.
Предложено множество различных механизмов представления концептуальных знаний. Чаще всего сейчас применяются следующие три.

- *Правила, или продукции*. ПСИИ, основанные на правилах, в настоящее время встречаются чаще всего. С математической точки зрения такие системы используют модели вычислимости, близкие к нормальным алгорифмам Маркова. Системы продукций детально рассматриваются во второй и третьей темах данного курса.
- *Формулы логических исчислений*. ПСИИ, основанные на логических формулах, в настоящее время занимают второе место по распространенности. С математической точки зрения такие системы используют исчисления, близкие к исчислению предикатов первого порядка. Представление знаний с помощью формул логических исчислений рассматривается в четвертой и пятой темах данного курса.
- *Фреймы и семантические сети*. В настоящее время этот механизм используется сравнительно редко, хотя в недавнем прошлом это было не так. С математической точки зрения такие системы используют аппарат теории графов. Представление знаний с помощью формул логических исчислений рассматривается на семинарских занятиях данного курса.

Все три распространенных механизма, вообще говоря, эквивалентны с точки зрения выразительной силы и других общематематических свойств. Предпочтение тому или иному механизму представления знаний часто отдается исходя из соображений привычности, удобства программирования, знакомства разработчиков ПСИИ с теорией.

1.3.3. Классификация по виду ответа при решении задач

Решая конкретную задачу, ПСИИ получает на вход знания в той или иной форме а на выходе выдает ответ, который также имеет некоторую форму и представляет собой некоторое (новое) знание. В соответствии с введенной классификацией видов знаний ПСИИ можно классифицировать по виду выдаваемого ответа:
а) логический ответ (да-нет);
б) фактографический ответ; ответ (факт); если выдается конкретный ответ на одну задачу, такие системы часто называют *вопрос-ответными системами*;
в) процедурный ответ; решая задачу, система может создать и запустить процедуру (система синтеза программ; автоматическое программирование);
г) ответ на понятийном уровне; ответ-закон, строится не решение, а схема решения класса задач, может быть даже на компьютере не выполнимая.
Примеры ответов разных типов (используется модельный пример с сортировкой чисел):

а) 2 5 7 → да
5 2 7 → нет
б) 5 2 7 → 2 5 7
в) $\{x, y, z\}?\{x < y < z\}$ → `Sort3`
г) $\{x_i\}$ **отсортировать** → $\forall\, i, j\ (\, i < j \rightarrow x_i < x_j\,)$

1.3.4. Классификация по степени универсальности

Вначале каждая новая идея рождается привязанной к предметной области. В дальнейшем возникает соблазн сделать СПЗ как можно более универсальной для решения практически любых задач. Однако стремление сделать СПЗ универсальной достаточно бесполезно, поскольку с ростом универсальности резко падает применимость таких СПЗ (рис. 7). Ни в БД, ни в ППП, ни в ПСИИ нет универсальности. Причины достаточно просты. Во-первых, от предметной области зависит способы манипулирования знаниями, во-вторых, слабы метазнания.

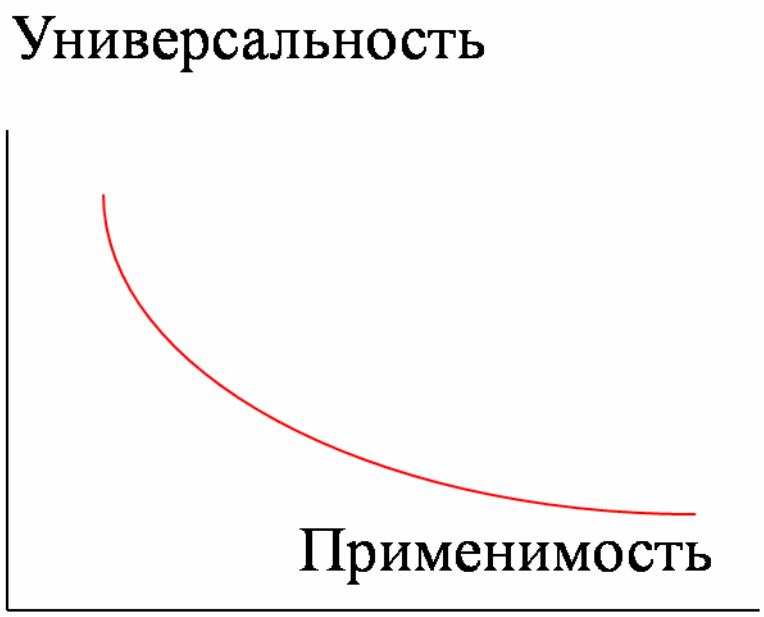


Рис. 7. Зависимость между универсальностью и применимостью

1.4. Архитектура ПСИИ

При анализе типичных ПСИИ логично выделить:

- *объекты*, с которыми манипулирует система (другими словами, что дано),
- *действия*, посредством которых эти манипуляции производятся (что можно сделать с этими данными), и
- *стратегию управления*, которая определяет когда и какие манипуляции следует производить (в каком порядке, к каким объектам, какие действия нужно применять).

Такое представление об архитектуре ПСИИ не догма – оно возникает при анализе *типичных* современных систем искусственного интеллекта. Узко специализированные системы иногда имеет другую архитектуру

Имеется виду логический анализ, а не анализ на уровне машинных кодов, где архитектурная структура уже неразличима.
Сейчас про эти три компонента (особенно если они четко разделены в программной реализации) говорят: данные (свойства объектов), алгоритмы (методы объектов) и метазнания.

Тема 2. Представление знаний системами продукций

Представление знаний системами продукций (Production Rules Knowledge Representation) является одним из наиболее часто используемых механизмов представления знаний в компьютере.

2.1. Системы продукций

Прикладные системы с элементами искусственного интеллекта ПСИИ, основанные на формализме продукций, имеют четкое деление на три упомянутые в предыдущей главе части – фактографическое знание (базы данных), алгоритмическое знание (правила), концептуальное знание (стратегия управления). Именно поэтому систем продукций (СП) так хорошо отражают ПСИИ в целом.

2.1.1. Структура систем продукций

Системы продукций (или продукционные системы, или системы, основанные на правилах) ([Production] [rules] [based] [inference] systems) имеют следующий состав:

СП = глобальная база данных + множество правил + система управления.

В этом учебнике база данных систематически обозначается буквой D, множество продукций – R, а стратегия управления – C.

$$\text{СП} = D + R + C\ [+\ t]$$

База данных D бывает разной сложности – сложность любой БД зависит от конкретных задач.

Множество правил R имеет вид:

$$R = \{r \mid r = (p, f\ [,c])\},\ r \in R,\ r = (p, f)$$

Каждое правило имеет вид

$$r = \textbf{if}\ p\ (D)\ \textbf{then}\ D := f\ (D)\ \textbf{end if}$$

Таким образом, применение правила r изменяет базу данных D.

Множество всех возможных состояний базы данных D называется *пространством поиска*.

Кроме этого, обычно существует *условие окончания* процесса поиска $t(D)$, то есть предикат, аргументом которого является база данных. Если условие $t(D)$ выполнено, то считается, больше применять продукции не нужно, и текущее состояние базы данных D является ответом.

Правила применяются в соответствии со стратегией управления, например до тех пор, пока либо не достигается условие окончания, либо не окажется больше применимых правил.

Организация ПСИИ в виде СП обладает одним очень важным преимуществом – модульностью или *алгоритмичностью* знания. Как следствие, такие системы способны обучаться, что является одним из самых важных критериев ИИ.

Отличие от иерархических (структурно) организованных программ заключается в следующем:

- Между правилами нет прямой связи по управлению. Иными словами, правила не вызывают друг друга. Все управление правилами вынесено в стратегию управления C.

- Между правилами нет и прямой связи по данным. Правила ничего не сообщают друг другу, все данные находятся в базе данных *D*.

Отсюда следует важный вывод – правила являются хорошо замкнутыми информационно прочными модулями.
В ПСИИ модульность сверхкритична, так как для программ ПСИИ очень часто не ясен алгоритм действия. А значит, нужна и сверхмодифицируемость, чтобы пробовать разные способы решения задачи.

2.1.2. Нормальные алгорифмы Маркова

На самом деле, нет резкой границы между ПСИИ и другим программным обеспечением. Системы продукций – это просто еще один способ организации вычислений. Этот способ совершенно не похож на привычное процедурное и объектно-ориентированное программирование, но обладает такими же возможностями.

Андрей Андреевич Марков (1903 —1979) — *советский математик, сын известного русского математика А. А. Маркова. Основные труды по теории динамических систем, топологии, топологической алгебре, теории алгоритмов и конструктивной математике.*
Доказал неразрешимость проблемы равенства в ассоциативных системах (1947)*, проблемы гомеоморфии в топологии* (1958)*, создал школу конструктивной математики и логики в СССР, автор понятия нормального алгорифма.*

Механизм системы продукций не является откровением, чем-то уникальным и особенным. Фактически СП основана на модели вычислимости, которая называется «нормальные алгорифмы[4] Маркова»:
Нормальный алгорифм Маркова работает со словами – конечными последовательностями символов конечного алфавита.
Нормальный алгорифм записывается как последовательность правил. Каждое правило имеет вид

```
слово → слово
```

где слово слева от стрелки называется левой частью правила, а слово справа от стрелки – правой частью. Или же имеет вид

```
слово →. слово
```

и в этом случае правило называется заключительным. Слова в левых и правых частях правил могут быть пустыми.

[4] Андрей Андреевич Марков принадлежал Московской математической школе, в которой принято написание «алгорифм». В соответствии с традицией, в обороте «нормальный алгорифм» мы оставляем букву «ф», хотя в остальных случаях используем слово «алгоритм», как это принято в современном русском языке.

Нормальный алгорифм выполняется следующим образом. Имеется исходное слово. Последовательность правил просматривается, начиная с первого правила, и проверяется, не входит ли левая часть правила в текущее слово. Если левая часть правила входит в текущее слово в качестве подслова, то самое левое такое вхождение левой части заменяется правой частью правила. Это называется применением правила. Если было применено заключительное правило, то на этом работа алгорифма заканчивается успешно: полученное в результате преобразований текущее слово является ответом. Если примененное правило не было заключительным, то работа алгорифма продолжается: снова рассматривается первое по порядку правило и ищется вхождение его левой части в текущее слово и так далее. Если же левая часть текущего правила не входит в текущее слово, то проверяется следующее по порядку правило. Если не оказалось применимых правил, то работа алгоритма заканчивается неудачей.

Рассмотрим пример нормального алгорифма, который прибавляет к числу в двоичной записи единицу.

```
X := X + 1
```

Запись числа состоит из символов 0 и 1. Добавим в алфавит еще две вспомогательные буквы: a и b Алгорифм состоит из восьми правил. Для удобства правила перенумерованы, эти номера не являются частью записи алгорифма.

```
1) a0 → 0a
2) a1 → 1a
3) 0a → 0b
4) 1a → 1b
5) 0b →. 1
6) 1b → b0
7) b  →. 1
8)    → a
```

Пусть теперь дано слово

```
101
```

Рассмотрим протокол работы алгорифма. В этом протоколе в каждой строчке слева от стрелки показано текущее слово, справа от стрелки показано слово, полученное в результате применения правила, а еще правее в скобках указан номер примененного правила.

```
101 → a101 (8)
a101 → 1a01 (2)
1a01 → 10a1 (1)
10a1 → 101a (2)
101a → 101b (4)
101b → 10b0 (6)
10b0 →. 110 (5)
```

Последним применилось пятое правило, оно является заключительным, процесс закончился и получен правильный ответ.
Несмотря на удивительную простоту нормальных алгорифмов Маркова доказано, что такое определение понятия вычислимости (алгоритма) эквивалентно всем другим, например машине Тьюринга или современным языкам программирования. Другими словами, в форме нормального алгорифма Маркова можно записать *любую* программу!

2.1.3. Основные процедуры

Рассмотрим простейший по результатам вид ПСИИ – только ответ да/нет, который соответствует тому, выполнено условие $t(D)$ или нет. При этом последовательность применения правил нам (может быть) неинтересна. Положительный ответ «да» здесь и далее в программах обозначается **OK**, отрицательный ответ «нет» – **fail**.

```
proc Production (D, R, t)
     while ¬t(D) do
          r := Select (D, R)
          if r = nil then
               return (fail)
          end if
          D := r.f (D)
     end while
     return (OK)
end proc
```

Функция `Select` – сердце этой системы, именно она определяет выбор применяемого правила, то есть стратегию системы продукций С.
Например, самая грубая стратегия поиска – первый подходящий:

```
proc Select (D, R)
     for r ∈ R do
          if p (D) then
               return (r)
          end if
     end for
     return (nil)
end proc
```

Это хороший пример плохой стратегии. Трудно ожидать интеллекта от машины, если мы не закладываем в нее немного своего.
Как правило, в ПСИИ недостаточно информации для того, чтобы на каждом шаге применять наилучшее правило. Поэтому процесс выбора правила есть процесс поиска.

2.2. Примеры

Эффективность ПСИИ определяется в первую очередь тем, насколько удачно выбрано ПЗ для всех компонентов *D*, *R*, *C*.

2.2.1. Игра в 8

Рассмотрим это на примере игры в 8, которая многим знакома по аналогии с известной игрой в 15. Суть игры заключается в перемещении фишек, пронумерованных от 1 до 8 так, чтобы они расположились по возрастанию по часовой стрелке. При этом одна из клеток поля 3×3 всегда свободна, и на нее можно перемещать любую соседнюю фишку. Фишки не могут «прыгать» друг через друга – их можно только передвигать на соседнее свободное место.
Допустим, у нас имеется следующая исходная позиция (рис. 8):

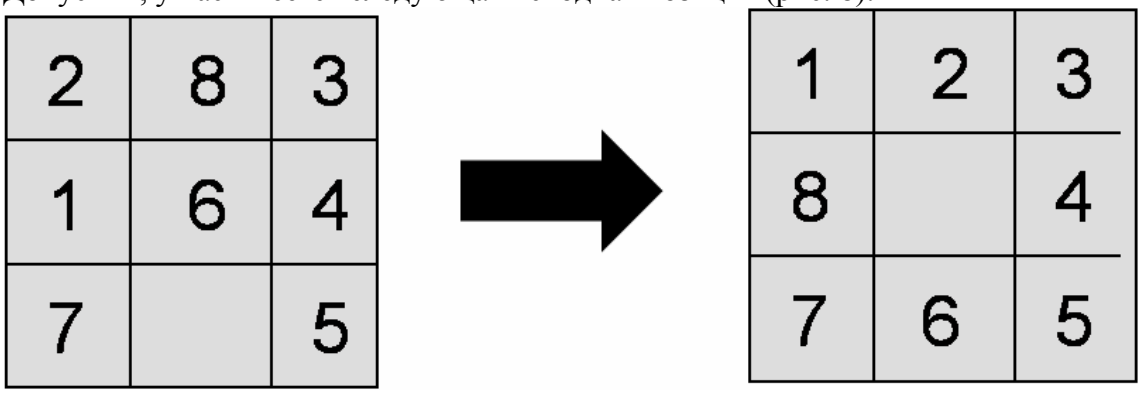


Рис. 8. Исходная и целевая позиции игры в 8

Целевой позицией будет являться та, что находится справа.
Попробуем составить алгоритм поиска решения на основе СП.
База данных напрашивается из физического смысла. Обозначим пустую ячейку как фишку с номером 0.
В результате *D* будет иметь вид:

```
D : array [1..3,1..3] of 0..8
```

Правила *R* тоже записываются достаточно ожидаемо. Например, правило «подвинуть фишку 1 вправо» имеет вид:

```
if  D[i,j]=1 & i<3 & D[i+1,j]=0 then D[i+1,j]:=1; D[i,j]:=0
```

Исходя из того, что любую фишку можно подвить в четыре стороны, получается всего 32 правила.
Это естественное, но не самое удачное представление знаний об игре!
Основная идея заключается в том, что любое перемещение фишки можно рассматривать как перемещение пустой ячейки. Вот здесь мы и закладываем немного своего интеллекта.
Добавим в базу данных еще пару чисел `{i,j}` – координаты пустого места.
Обозначим направления движения по сторонам света – север, запад, юг и восток.

```
if i<>1 then  D[i,j]:=D[i-1,j]; i:=i-1   N (движение на север)
if j<>3 then  D[i,j]:=D[i,j+1]; j:=j+1   E (движение на восток)
if i<>3 then  D[i,j]:=D[i+1,j]; i:=i+1   S (движение на юг)
if j<>1 then  D[i,j]:=D[i,j-1]; j:=j-1   W (движение на запад)
```

В результате составлено ПЗ для игры в 8, в котором всего 4 правила вместо 32. Это принципиальный результат, потому что число правил n (4 или 32) является аргументом экспоненциальной функции перебора для поиска решения. Для $n = 4$ мощности среднего компьютера наверняка хватит, а для $n = 32$ может и не хватить.

2.2.2. Крестьянин, волк, коза и капуста

Второй пример – известная задача о крестьянине, которому необходимо перевезти на другой берег волка, козу и капусту так, чтобы они не съели друг друга (рис. 9). В лодку крестьянин может взять с собой не более одного объекта.

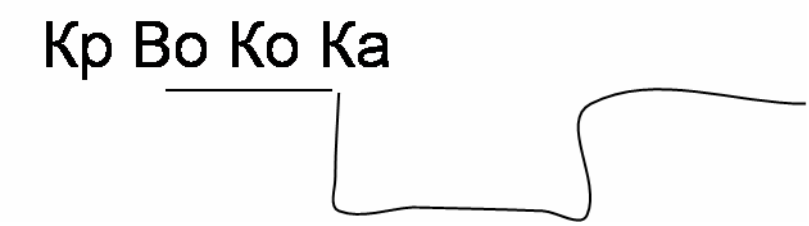


Рис. 9. Крестьянин, волк, коза и капуста: постановка задачи

Естественно, можно упростить данную задачу. Идея такова – крестьянин и лодка неразличимы, и составляют одно целое. Также можно считать любое перемещение с берега на берег мгновенным. Перенумеруем объекты: 0 – крестьянин, …, 3 – капуста. Присутствие объекта на правом берегу обозначим «1», на левом – «0».
Очевидно, что база данных D должна иметь вид

```
D : array [0..3] of 0..1
```

Тогда исходное состояние базы `D0 = 0000`, а целевое `Dk = 1111`

Каково должно быть множество правил? Если подойти к решению этой задачи прямолинейно, в лоб, то можно рассуждать следующим образом. Всего имеется 16 возможных состояний базы. Далее, рассмотрим пары состояний базы, такие, что из первого состояние можно перейти во второе за один ход. Например, `1011`→`0011` – крестьянин переплыл с правого берега на левый. Заметим, что некоторые состояния базы допустимы, например, `0101`, а другие состояния недопустимы, например, `0011` – коза съест капусту. Отбросим те пары состояний, в которых целевое состояние является недопустимым. Оставшиеся пары являются правилами нашей системы.
Нетрудно видеть, что это чисто механическая, формальная работа, которую можно поручить компьютеру. Однако результат окажется не очень хорошим (десятки правил).
Такое представление знаний нельзя назвать удачным.
Попробуем подойти к задаче творчески. В данном случае творчество состоит в привлечении знаний, которые в исходной постановке задачи не содержались.

1. Для решения этой задачи достаточно рассматривать только допустимые состояния базы (недопустимые не могут войти в решение даже на промежуточных этапах).

2. Если начальное состояние допустимо и все правила сохраняют допустимость, то любое достижимое состояние допустимо (принцип полной индукции).

Заметим, что начальное состояние допустимо (так же, как и конечное). Осталось выделить правила, сохраняющие допустимость. Для этого заметим, что в задаче, по существу, есть всего четыре возможных хода: движение крестьянина без объекта, движение крестьянина с волком, с козой и с капустой. Теперь осталось выписать простые условия допустимости этих ходов и получить множество правил *R*:

```
if D[1]<>D[2] & D[2]<>D[3]     then     D[0]:= 1 - D[0]
if D[2]<>D[3] & D[0]=D[1]      then     D[0]:= 1 - D[0];
                                        D[1]:= 1 - D[1]
if D[0]=D[2]                   then     D[0]:= 1 - D[0];
                                        D[2]:= 1 - D[2]
if D[1]<>D[2] & D[0]=D[3]      then     D[0]:= 1 - D[0];
                                        D[3]:= 1 - D[3]
```

2.3. Стратегии управления

Можно предложить огромное множество стратегий управления (Control Strategies) СП. Самое важное наблюдение состоит в том, что не существует и не может существовать наилучшей стратегии управления СП на все случае жизни. Одна и та же стратегия может хорошо работать для одного класса задач и быть совершенно неудовлетворительной для другого класса задач.

Нужны критерии для сравнения различных стратегий.

Некоторые варианты стратегий управления приведены на рисунке 10. Здесь используются два критерия:

- какая доля информации, уже полученной в ходе поиска решения сохраняется для последующего использования;
- до какой степени разрешены возвраты, то перевод базы данных не в новое, а в одно из старых состояний.

Заметим, что на самом деле эти критерии не являются вполне независимыми.

	Без возвратов	**С возвратами**
Без сохранения информации	• Первый подходящий	Поиск с • возвратами
С сохранением информации		Поиск на графе •

Рис. 10. Варианты стратегии управления.

Очевидно, что стратегия «первый подходящий» приведенная ранее, простейшая и соответствует левому верхнему углу на плоскости стратегий на рис. 10. Эта стратегия относится к группе стратегий «безвозвратный поиск».

Кроме безвозвратного поиска, здесь рассматриваются еще две популярные группы стратегий.

- Поиск на графе – сохраняется вся доступная информация и возможны любые возвраты, хоть в начальное состояние.
- Поиск с возвратами – промежуточный вариант, когда некоторая информация сохраняется, но не вся, и используются возвраты, но не произвольные.

Любую стратегию можно оценить по двум критериям – по затратам на выбор правил и по затратам на применение правил. (рис. 11). При этом, естественно, возможно два крайних случая – тупая, но быстрая стратегия и умная, но медленная стратегия. Как всегда, есть и золотая середина, где общая трудоемкость процесса будет минимальной.

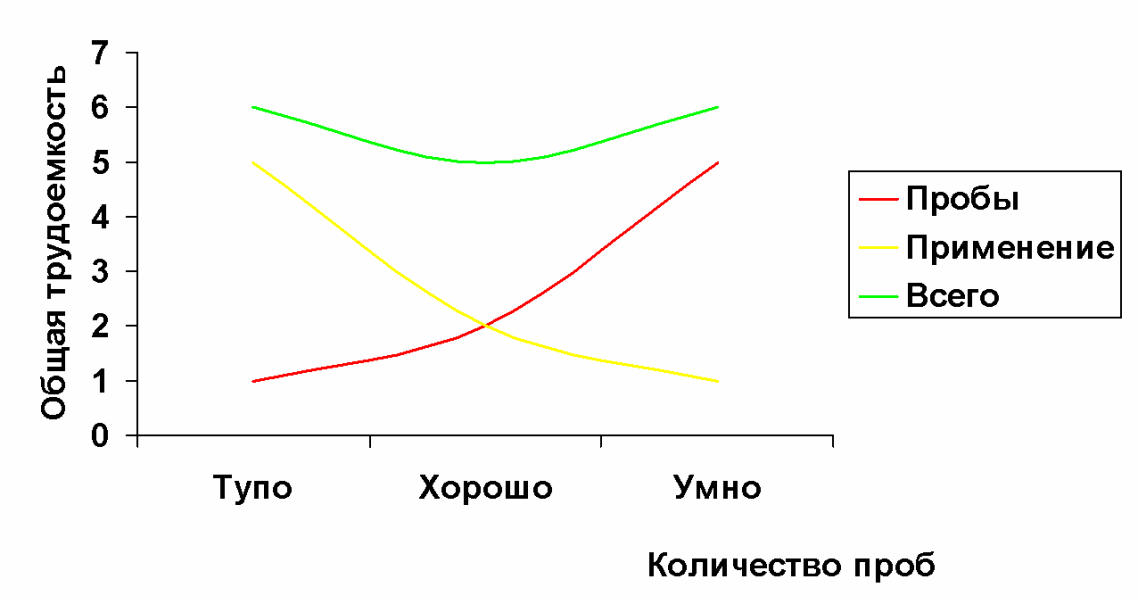


Рис. 11. Составляющие затрат на применение стратегии управления.

2.4. Безвозвратный поиск

Безвозвратный поиск, как следует из его названия, – это такой алгоритм поиска, когда мы не собираемся возвращаться к предыдущим состояниям базы данных, а значит, можем не хранить информацию об этих состояниях.

Простейшим случаем безвозвратного поиска является стратегия «первый подходящий», описанная выше. В этом случае поиск ведется вслепую, наудачу, которая существенным образом зависит от того, в каком порядке перебираются (а значит, и применяются) правила при проверке. Если мы ничего не знаем о задаче, а пространство поиска велико, то не стоит рассчитывать на то, что стратегия «первый подходящий» даст хороший результат.

2.4.1. Оценочная функция

В случае безвозвратного поиска вводится оценочная функция $F(D)$ – оценка состояния базы данных D. Функция подбирается таким образом, чтобы на терминальных узлах (тех состояниях базы, где выполняется условие окончания) функция принимала минимальное значение

$$t(D_k) := F(D_k) = \min F(D)$$

В этом случае функция `Select`, реализующая стратегию «первый подходящий, уменьшающий оценку», может быть записана следующим образом.

```
proc Select (D, R)
     for r ∈ R do
     if r.p(D) & F(r.f(D)) ≤ F(D) then
          return (r)
     end for
     return (nil)
end proc
```

2.4.2. Метод наискорейшего спуска

Можно улучшить процедуру, выбирая такое правило, которое не просто уменьшает оценку, а уменьшает ее в наибольшей степени.

```
proc Select (D, R)
     m := F(D)
     r' := nil
     for r ∈ R do
          if r.p (D) & F(r.f (D)) < m then
               r' := r
               m := F(r.f (D))
          end if
     end for
     return (r')
end proc
```

Такую стратегию часто называют *методом наискорейшего списка*, который относится к классу *жадных алгоритмов* (Greedy Search).

Как хорошо известно из вычислительной математики, у методов этого типа есть существенный недостаток. Иногда на пути решения возможны попадания в локальные минимумы и плато, которые в общем случае достаточно трудно обнаружить. Для этого требуются очень дорогие вычисления. Попав в локальный минимум, метод наискорейшего спуска остановится в нем, не найдя решения. Попав на плато, метод приведет к бесцельным случайным блужданиям по плато.

Метод наискорейшего спуска хорошо работает, если оценочная функция унимодальна. Но чтобы подобрать унимодальную оценочную функцию, нужно исчерпывающим образом знать свойства пространства поиска, а они, как правило, неизвестны.

Рассмотрим пример. В случае игры в 8 возьмем оценочную функцию F – это число шашек, находящихся не на своем месте. В результате дерево поиска этой игры будет выглядеть следующим образом (рис. 12).

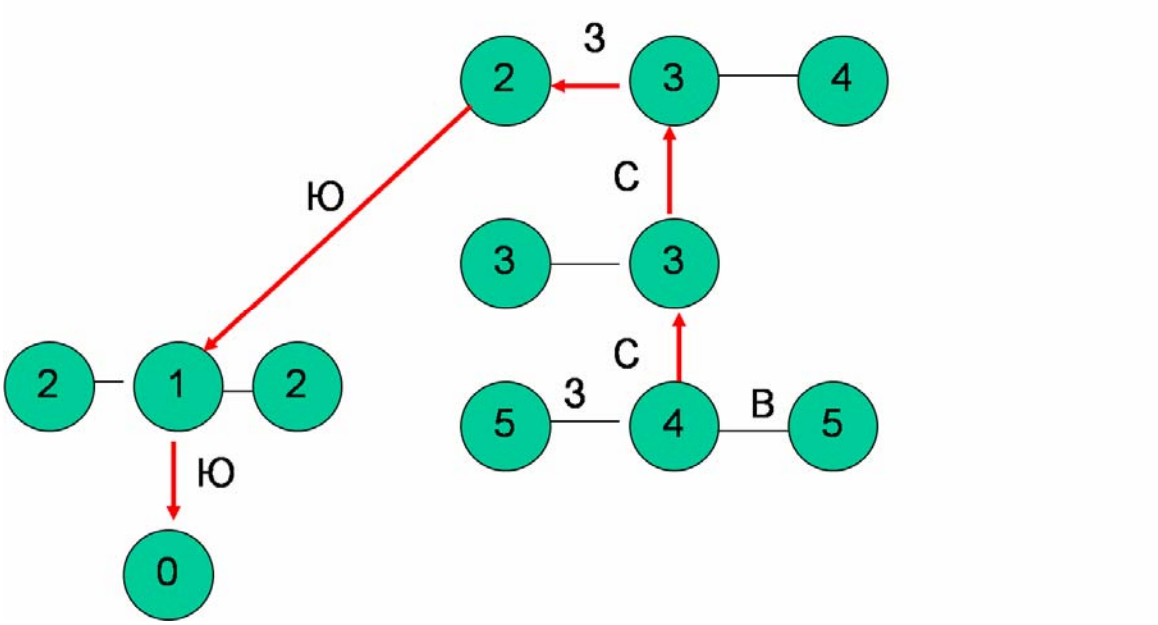


Рис. 12. Оценочная функция *F*: число шашек не на своем месте.

Здесь кружки обозначают состояния базы данных, а числа в кружках – это значение оценочной функции в данном состоянии.

В данном случае оценочная функция оказалась удачной – метод наискорейшего поиска быстро нашел решение. Но здесь есть элемент везения: из узла с оценкой 3 поиск пошел на север, а не на запад, просто потому, что правило движения на север в нашем списке оказалось раньше правила движения на запад.

2.5. Поиск с возвратами

Безвозвратный поиск подразумевает, что в случае применения неподходящего правила, решение либо становится принципиально невозможным, либо процесс поиска значительно замедляется.

Можно положить в основу алгоритма поиска следующую идею: если в текущем состоянии базы данных видно, что продолжая начатое, решение либо невозможно найти, либо этот поиск обещает быть слишком долгим, можно попробовать вернуться назад и попробовать другой пусть.

Стратегия поиска, в которой если применение правила оказывается неподходящим, то происходит возвращение назад и применение другого правила, называется *поиск с возвратами*[5] (Backtracking).

Рассмотрим процедуру `Backtracking`.

```
proc Backtracking (D, R, t)
          if t(D) then return (OK) end if
          if Deadend (D) then return (fail) end if
          R' := R
M:    r := Select (D, R')
```

[5] Иногда применяют неудачный термин «обратное прослеживание». Этот термин является точным переводом с английского, но плохо отражает суть дела.

```
    if r = nil then return (fail) end if
    R' := R' \ r
    if Backtracking (r.f(D), R, t) = fail then goto M
        else return (OK) end if
end proc
```

В данную процедуру необходимо включить функцию `Deadend` – обрыв «мертвых» путей (тупиков), иначе может произойти зацикливание программы.
Процедура `Deadend` может проверять самые разные условия, например:

- совпадение *D* с уже построенным ранее;
- ограничение глубины поиска и т.д.

Известны различные модификации поиска с возвратами, например, когда в процессе поиска можно сохраняются не все, а только ключевые состояния базы данных.

Пример. Поиск с возвратами для задачи о волке, козе и капусте представлен на рис. 13.

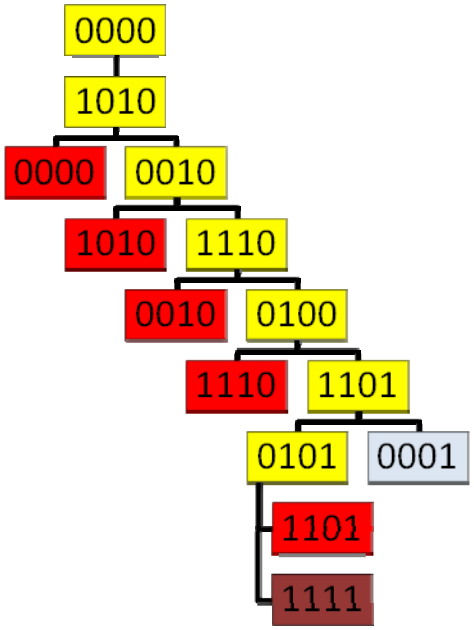


Рис. 13. Поиск возвратами

Здесь в качестве функции `Deadend` используется проверка совпадения нового состояния базы с построенным ранее на этом пути.

2.6. Поиск на графе

Поиск на графе (Graph Search) относится к группе информированных стратегий, когда хранится вся доступная полученная ранее информация.

2.6.1. Основные процедуры

Для поиска на графе используются следующие процедуры.

```
proc GraphSearch (D, R, t)
          O := {D} { список «открытых» узлов }
```

```
          C := ∅  { список «закрытых» узлов }
M:   D := Select (O)
          if D = nil then return (fail) end if
          if t(D) then return (OK) end if
          Open (D);
          goto M
end proc

proc Open (D)
     for r ∈ R do
          if r.p (D) then
               if r.f(D) ∉ O & r.f(D) ∉ C then
                    O := O + r.f (D)
               end if
          end if
     end for
     O := O - {D}
     C := C + {D}
end proc
```

Здесь знак + означает добавление элемента в список, а знак – означает удаление элемента из списка.

Список *O* традиционно называется списком открытых узлов (от английского Open), а список *C* традиционно называется списком закрытых узлов (от английского Closed). Между тем эти традиционные названия могут вводить в заблуждение. На самом деле список *O* – это список узлов, которые мы еще не открывали, но собираемся открыть (to be opened), а список *C* – это список узлов, которые мы уже открыли, и больше открывать не будем (already opened).

2.6.2. Граф поиска

В случае поиска на графе создается так называемый граф поиска.

Граф поиска – это ориентированный граф $G(V, E)$, в котором множество узлов V – это множество состояний базы данных, а множество дуг E – это множество примененных правил.

По своей сути узлы – это состояния БД. Кроме этого, есть множество дуг. Дуги – это правила.

$$D_1 \to D_2 \in E \leftrightarrow \exists\, r \in R\ (\, r.p(D_1)\ \&\ r.f\,(D_1) = D_2\,).$$

В этом графе есть начальный узел D_0 и множество (возможно, пустое) терминальных вершин $T = \{D \mid t(D)\}$. Элементы множества терминальных узлов будем обозначать D_t.

В общем случае в графе поиска могут быть циклы, петли, мультидуги. Он может быть не связан и т.д.

Пример. При игре в 8 очевидно, что всего существует 8!9 = 9! состояний базы данных (число перестановок фишек, умноженное на число положений пустой ячейки). Можно доказать, что все состояния БД разбиваются на два множества: с четным числом инверсий и с нечетным числом инверсий, причем ходы обязательно сохраняют четность. Таким образом, граф поиска для игры в 8 состоит из двух компонент связности, причем каждая компонента сильно связна. Если начальное и целевое состояния базы принадлежат разным компонентам связности, то решение невозможно, а если принадлежат одной компоненте, то решение обязательно существует.

Мы видим, что в общем случае конечное множество правил R неявно задает, возможно, бесконечный граф поиска.

В терминах графа поиска можно сказать, что поиск с возвратами – это щуп, который мы «засовываем» в граф поиска в надежде наткнуться на D_t. В результате происходит проявление части графа.

2.6.3. Поиск в ширину и в глубину

Существуют два принципиальных способа поиска на графе – поиск в ширину и поиск в глубину.

Если список O – это стек, то есть для раскрытия выбирается последний положенный узел, то в этом случае говорят о *поиске в глубину*.

Если список O – это очередь, то есть для раскрытия выбирается первый положенный узел, то в этом случае говорят о *поиске в ширину*.

У поисков в ширину и глубину есть свои достоинства и недостатки.

При решении любой задачи всегда существует дилемма – проверять или не проверять дублирование узлов?

Если дублирование проверять, то поиск в ширину может построить меньшее количество узлов, чем поиск в глубину.

Если граф конечный, то поиск в ширину и поиск в глубины полны, то есть найдут путь, если он есть. Но в случае бесконечного графа, поиск в ширину найдет путь, если он есть, но искать его будет достаточно долго; поиск в глубину найдет быстрее, но при этом может пройти мимо нужного пути и уйти в бесконечность.

Пример. Поиск в ширину для задаче о крестьянине, волке, козе и капусте (рис. 14).

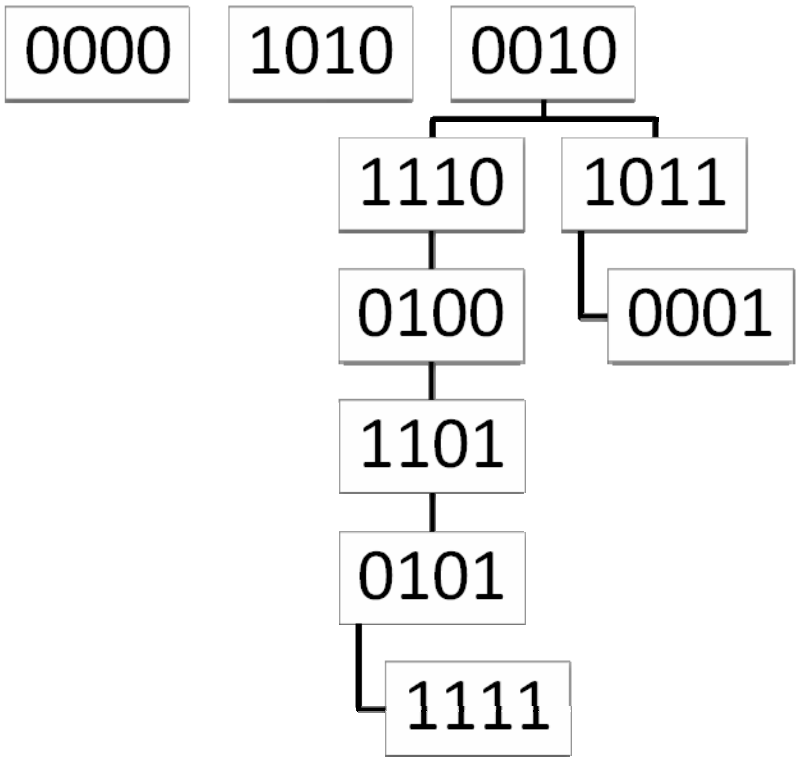


Рис. 14. Поиск в ширину на графе

Заметим важное обстоятельство. Вопреки распространенному предрассудку, поиск в ширину может строить меньше узлов, чем поиск с возвратами, если проверять совпадения (сравните рис. 13 и рис. 14). Другими словами, поиск в ширину на графе (полный перебор) может в некоторых случаях быть эффективнее поиска с возвратами!

2.7. Обратные и двусторонние системы продукций

Рассмотренные СП всегда начинали работать от исходного состояния до тех пор, пока в результате не получалось целевое состояние. Такие СП называются прямыми. Логично поступать и наоборот – применять обратные правила и двигаться от целевого состояния к исходному.

Возможны два случая – когда обратная система лучше или когда лучше прямая система (см. рис. 15, слева). Такие системы являются односторонними.

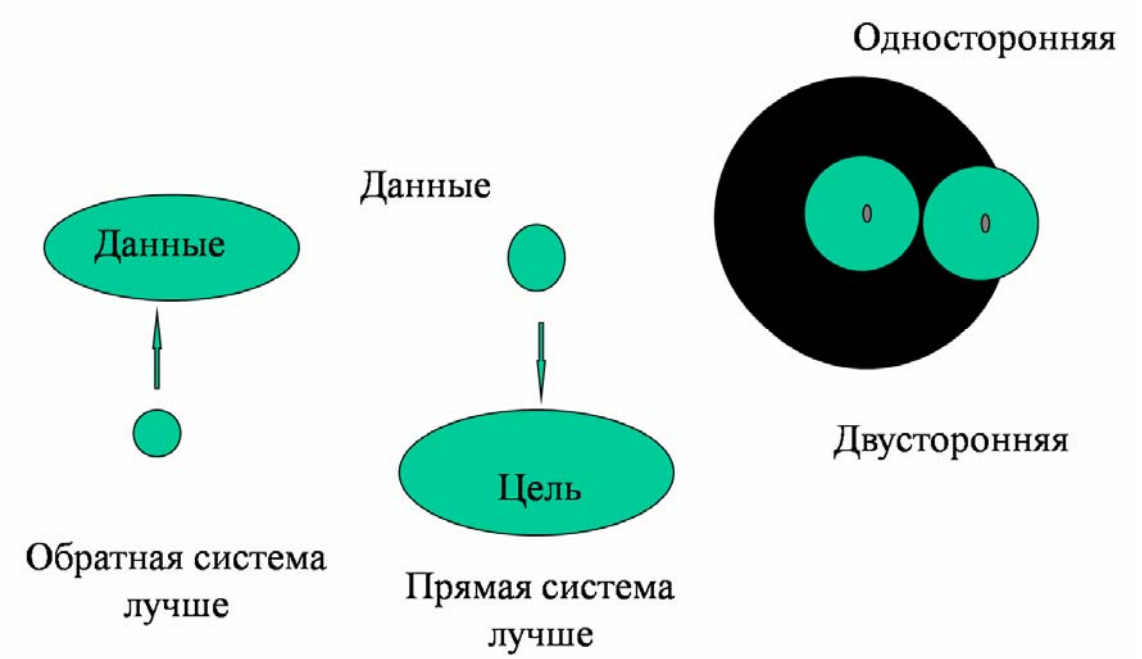


Рис. 15. Обратные и двусторонние СП

Если существует только одно целевое состояние и одно исходное состояние, то направление эквивалентны (например, при игре в 8). Это типичные двусторонние системы. В таком случае можно одновременно двигаться от данных к цели и от цели к данным, проверяя, не появилось ли общее состояние, которое свидетельствует о том, что решение найдено.

Можно ожидать, что если используется малоинформированная стратегия управления, то двусторонняя СП окажется эффективнее односторонней, поскольку суммарно меньше узлов будет раскрыто, например, при поиске в ширину (см. рис. 15, справа).

Однако это не всегда так. Если используются очень хорошие эвристики, которые целенаправленно ведут от данных к цели и от цели к данным, но при этом разными путями, то может оказаться, что двусторонняя система хуже, чем любая односторонняя (рис. 16)!

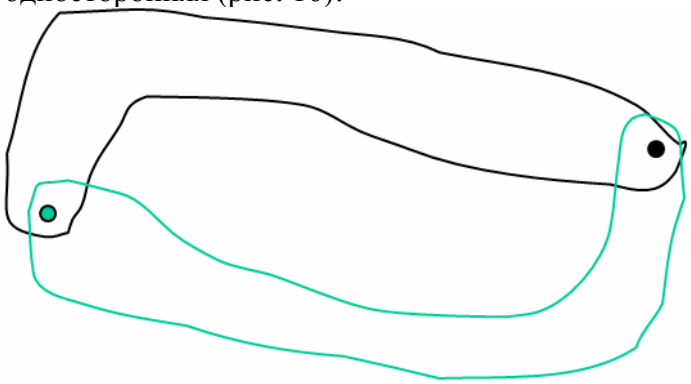


Рис. 16. Двусторонняя система с направленными эвристиками

Примеры. Рассмотрим следующую модельную задачу из теории чисел. Нужно определить, разлагается ли заданное число на заданные множители.

Пусть целевое число равно 60, а множители – 2, 3, 5.

Прямая система:

```
D0 := {2,3,5}, t(D) := 60 ∈ D
R := if x∈D & y∈D & xy∉D & xy<61 then D:= D∪{xy}
```

Поиск в ширину дает следующие результаты (10 применений):
2 3 5
6 10 15
12 20 30 18 45 50 60

Поиск в глубину дает следующие результаты (7 применений):
2 3 5 15 45 30 60

Обратная система:

```
D0 := {60}, t(D) := 1∈D
R := if x∈D & x:2 then D := D\{x}∪{x/2}
     if x∈D & x:3 then D := D\{x}∪{x/3}
     if x∈D & x:5 then D := D\{x}∪{x/5}
```

Поиск в глубину дает следующие результаты (4 применения):
60 30 15 5 1

Поиск в ширину следующие результаты (10 применений)
60
30 20 12
15 10 6 4
5 3 2
1

Здесь цель – единственная, а данных – три, поэтому, как и следовало ожидать, обратная система дает лучшие результаты.

Но если поставить другой вопрос: какие числа, меньшие или равные 60, можно получить из множителей 2, 3, 5, то прямая система, очевидно, окажется более эффективной.

2.8. Коммутативные системы продукций

Система продукций с множеством правил R называется *коммутативной*, если выполнены следующие условия:

$\forall D\ (\ r1.p\ (D)\ \&\ r2.p\ (D) \to .\ r2.p\ (r1.f\ (D))\ )$ (если правило применимо, то навсегда)

$\forall D\ (\ r.p(D)\ \&\ t(D) \to t(r.f(D))\ )$ (если цель достигнута, то насовсем)

$\forall D\ r1.p(D)\ \&\ r2.p(D) \to .\ r2.f(r1.f(D)) = r1.f(r2.f(D))$ (если правила применимы, то они коммутируют)

Пример. Рассмотренная в предыдущем разделе СП для чисел является коммутативной. Другой пример коммутативной СП – игра в 8, а вот задача о волке, козе капусте – это некоммутативная система.
Заметим, что следующее утверждения неверно:

$$\forall D\ r2.f(r1.f(\mathrm{D})) = r1.f(r2.f(\mathrm{D})),$$

то есть нельзя произвольно переупорядочить последовательность правил, но множество применимых в данный момент правил можно применять в любом порядке.
Одним из важных достоинств коммутативной СП – это возможность всегда использовать безвозвратный режим, так как применение правила ничего не портит.
Теорема. *Всякую СП можно преобразовать в коммутативную*.
Доказательство. Пусть задана СП $S = D + R + C$ с начальным состоянием базы D_0. Рассмотрим ее граф поиска G. Построим новую СП $S' = D' + R' + C'$. Базой данных СП S' будет граф поиска СП S. Правила R' новой системы – это различные способы преобразования графа поиска, которое можно делать с помощью стратегии C в старой системе (раскрытие узла). Новая СП коммутативна. Действительно:

- Если в старой системе два узла находятся в списке O, то раскрытие одного узла не препятствует раскрытию второго узла, так что первое условие коммутативности выполнено.
- Если в старой системе в графе поиска построен терминальный узел, то раскрытие других узлов не исключит из графа терминального узла, так что второе условие коммутативности выполнено.
- Если в старой системе два узла находятся в списке O, то их можно раскрывать в любом порядке, получится один и тот же результат, так что третье условие коммутативности выполнено.

Ч.т.д.
Доказанная теорема мало что дает на практике. Хотя полученная новая система коммутативна, но преобразование резко увеличивает ее объем по сравнению с исходной. Выигрыш за счет применения безвозвратного поиска оказывается меньше, чем проигрыш за счет разрастания систем при преобразовании.
Пример. Рассмотрим пример коммутативной системы для упрощения алгебраических выражений, построенных из переменной x, натуральных коэффициентов, операций сложения, умножения и возведения в натуральную степень (полиномы одной переменной). Попытаемся упростить выражение $1 + x(1+x(3+x+x))$ (рис. 17).
Имеется набор правил:

1. Раскрыть скобки
2. Привести подобные
3. Упорядочить по степеням

Эти правила всем известны со школы, поэтому мы не стали записывать их формально.

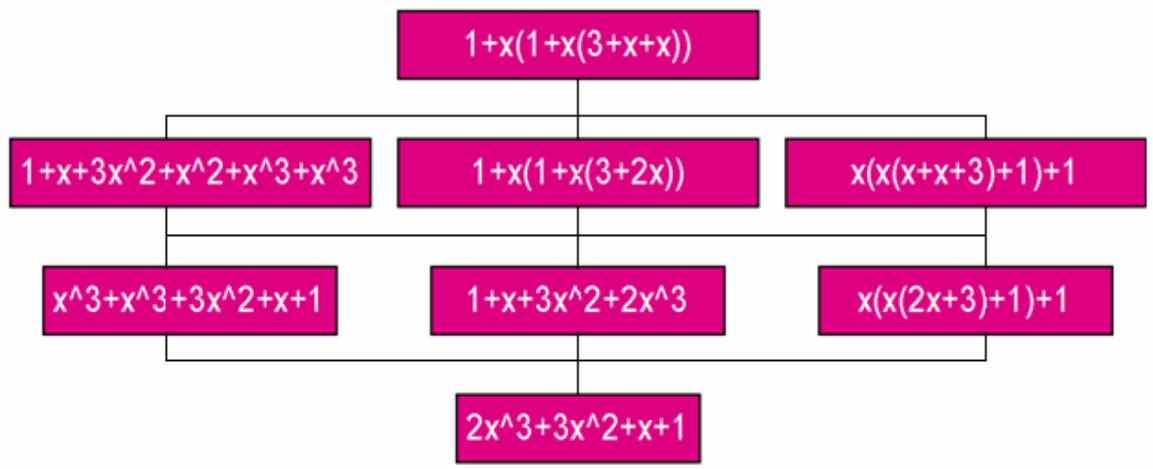


Рис. 17. Упрощение выражений.

Если правило что-то меняет, применяем его. Для полиномов окончательный результат не зависит от порядка применения правил. Фактически здесь СП не содержала условий – это пример *системы подстановки термов* (СПТ), системы правил переписывания (terms rewriting rules).

Система подстановки термов обладает *свойством Чёрча-Россера,* если для любой исходной формулы результат применения правил в произвольном порядке существует и единственен. Он называется *канонической формой*.

В данном случае система обладает свойством Чёрча-Россера. Это очень сильное, полезное и редкое свойство. Наличие свойства Чёрча-Россера позволяет легко решить автоматически многие задачи, такие как проверка равенств, решение уравнений и т.д. Однако канонические формы существуют не для всех алгебраических выражений. Существуют и известны читателю канонические формы для полиномов одной переменной и для булевых формул. В то же время, например, для элементарных функций математического анализа (дробно-рациональные функции, логарифм, экспонента и их суперпозиции) нормальной формы не существует.[6]

[6] Это объясняет, почему задачи, предлагаемые на экзаменах по элементарной математике («решить уравнение», «упростить выражение»), действительно иногда являются задачами, которые для своего решения требуют изобретательности и фантазии, а не только механического применения заученных правил.

Алонзо Чёрч (*Alonzo Church*), 1903 – 1995, *американский математик и логик, внесший вклад в основы информатики. Прославился разработкой теории лямбда-исчислений, последовавшей за его знаменитой статьёй* 1936 *года, в которой он показал существование т. н. «неразрешимых задач».*
Помимо прочего, его система лямбда-исчислений легла в основу функциональных языков программирования, в частности семейства Лисп (*например, Scheme*).

Теперь к нашему примеру добавим деление. Попытаемся упростить выражение $(x^2+3x+2) / (x+2)$, которое в результате должно превратиться в $x+1$.
Для этого добавим еще два правила:

4. Привести к общему знаменателю
5. Сократить числитель и знаменатель

Легко видеть, что результат зависит от порядка применения правил! Более того, в некоторых случаях возможны тупики в поиске. В результате один из путей не редуцируется правилами 1-5 (рис. 18).

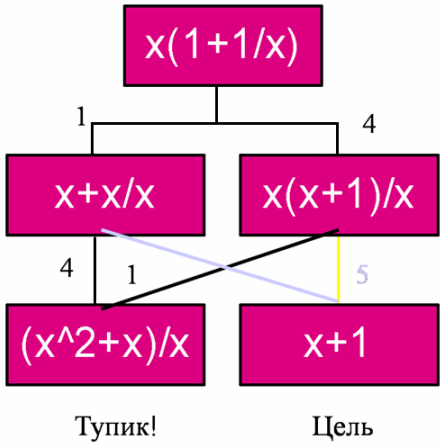


Рис. 18. Тупики при поиске

Для решения этой проблемы, связанной с образованием тупиковых ветвей, добавим обратные операции:

6. Вынести за скобки

Если ввести правило 6 (вынесение за скобки), то будут два правила: раскрытие скобок и вынесение за скобки, а это значит, что возможно зацикливание, то есть свойства Черча-Россера не выполняются по-прежнему. Чтобы это решение сделать редуцируемым, необходимо добавить правило 7:

7. Разложить дробь

$(u+v)/w = u/w + v/w$ (рис. 19)

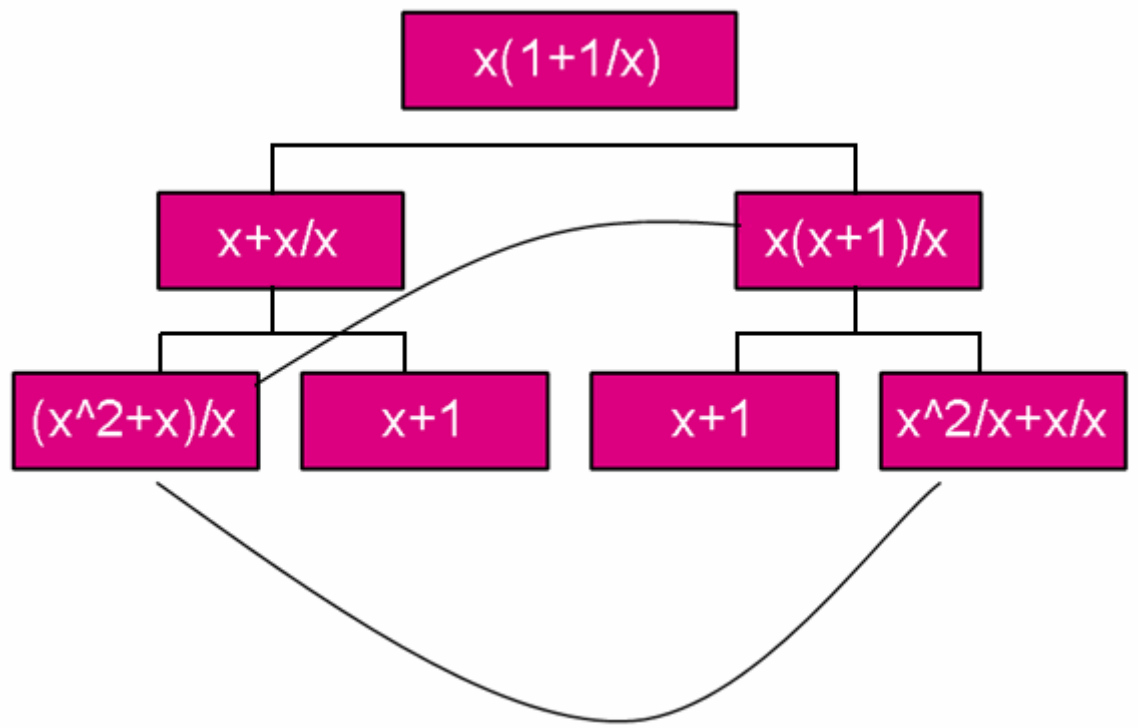


Рис. 19. Циклы при поиске

Но тогда может появиться бесконечное зацикливание, такое, которое при безвозвратном поиске невозможно распознать.

Вывод по данному примеру: коммутативность системы продукций – очень мощное и полезное свойство. Нужно стремиться строить коммутативные системы продукций, а в тех случаях, когда это невозможно, стоит проверить, нельзя ли немного ограничить задачу, но так, чтобы стало возможным построить коммутативную систему.

2.9. Разложимые системы продукций

Коммутативность СП предоставляет свободу выбора применяемого правила. Это не единственное свойство, которое дает такую свободу.

2.9.1. Свойство разложимости

Если исходная СП может быть разложена (Splitting) в сумму СП, которые могут рассматриваться независимо, то такую систему называют *разложимой системой продукций*.

Пример. Хороший пример разложимой СП дают формальные порождающие грамматики. Пусть имеются правила

A → BC

Ą →CT

Ç →TT

Вопрос к СП: Можно ли из слова AC получить цепочку, содержащую только символ T? Ниже представлены графы поиска для обычной (рис. 20) и разложимой (рис. 21) СП.

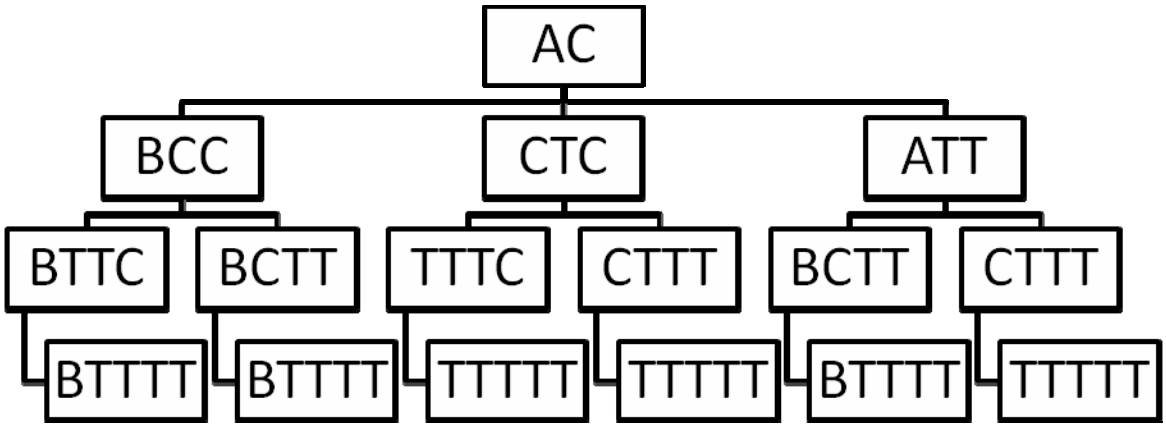


Рис. 20. Граф поиска для обычной системы продукций.

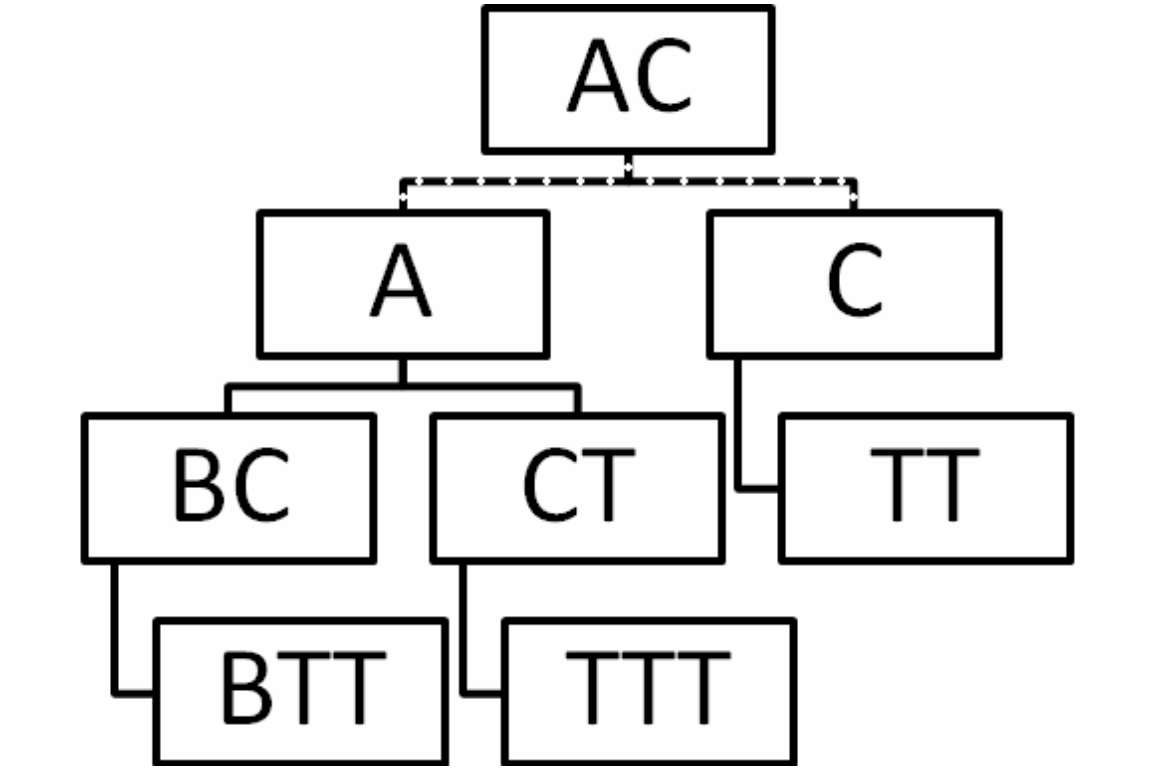


Рис. 21. Граф поиска для разложимой СП.

Как можно видеть из рисунка 20, в этом случае проделывается много лишней работы: в левой тупиковой ветви применялись только нужные правила.

2.9.2. Основная процедура для разложимой СП

```
proc Split (D, R, t)
```

```
            Δ := Decompose (D)
M:          for D∈Δ do
                 if t (D) then Δ := Δ \ {D} end if
            end for
            if Δ = ∅ then return (OK) end if
            D := SelectD (Δ)
            Δ:= Δ \ {D}
            r := Select (D, R)
            if r = nil then return (fail) end if
             Δ := Δ ∪ Decompose (r.f (D))
            goto M
end proc
```

Это аналог процедуры `Production`, самой первой и общей. Процедура `Decompose` разлагает базу данных на несколько частей (или оставляет без изменения, если в данном состоянии базу разложить невозможно). В известном смысле это похоже на операцию раскрытия узла, поэтому на рис. 21 дуги, соответствующие разложению, а не раскрытию, выделены пунктиром. Далее, функция `SelectD` аналогична функции `Select`, она фактически выбирает узел для дальнейшей работы.

Сделаем важное замечание. Фактически, с точки зрения теории графов, та дуга, которая появляется в графе поиска в результате применения правила разложения являются гипердугой: эта дуги исходят из разлагаемой базы и ведет сразу в несколько баз, являющихся частями разложения. Для того, чтобы найти решение, нужно найти решение для первой части И для второй части И так далее. Поэтому такая гипердуга называется И-дугой. Если же из зла выходят обычные дуги, соответствующие применению правил, то они имеют следующий смысл: чтобы найти решение нужно применить первое правило ИЛИ второе правило ИЛИ и так далее. Поэтому такие дуги называются ИЛИ-дугами.

Тема 3. Алгоритмы поиска

В этой теме более детально исследуются свойства поиска на графе, как наиболее информированной стратегии поиска.

3.1. Общий алгоритм

Для уточнения дальнейшего изложения повторим определение графа поиска и сделаем несколько важных замечаний.

Граф поиска – это ориентированный граф $G(V, E)$, в котором множество узлов $V = \{ D \mid D \in V \}$ – это множество возможных состояний базы данных, а множество дуг E определяется следующим образом:

$$E = \{ D \to D' \mid D, D' \in V \;\&\; \exists\, r \in R \; (r.p(D) \;\&\; D' = r.f(D)) \} \; .$$

В случае, когда граф поиска существенно конечен, его можно задать явно с помощью обычных способов представления графов в компьютере. В этом случае (когда граф конечен), как правило, именно граф поиска выбирается в качестве механизма представления знаний – его дуги являются правилами, отдельно задавать множество правил нет необходимости.

В случае бесконечного (или очень большого) графа мы не можем использовать указанное простое представление, просто потому, что весь граф не помещается в памяти компьютера. Вместо это мы, начиная с некоторого исходного состояния базы данных D_0, с помощью конечного множества правил R **порождаем** некоторую **часть** графа поиска, а именно ту, которая содержит целевое состояние базы данных D_t. Таким образом, используется неявное представление графа в компьютере. Неявно граф может быть задан с помощью операции раскрытия узла.

Очень часто каждому правилу можно сопоставить стоимость его применения $r.c$. В этом случае, в графе поиска каждая дуга будет иметь стоимость, и значит можно определить стоимость пути. Другими словами, дуги нагружены (*цена* или *длина*):

$$r = (p, f, c).$$

Общий алгоритм поиска на графе, который вычисляет стоимость путей от исходной вершины и определяет путь минимальной стоимости, выглядит следующим образом.

```
proc GS (D0, R, t)
     O := {D0};      g(D0) := 0
     C := ∅;         P(D0) := nil
M:   D := Select (O)
     if D = nil then return (fail) end if
     if t(D) then return (OK) end if
     Open (D)
     goto M
end proc
```

```
proc Open (D)
for r ∈ R do
          if r.p (D) then { правило применимо }
               D′ := r.f(D)
               c′ := g(D)+r.c
               if D′ ∈ O ∨ D′ ∈ C then { старый узел}
                    if c′ < g(D′) then
                         g(D′) := c′
                         P(D′) := D
                    end if
               else { новый узел }
                    O := O+{D′}
                    g(D′) := c′
                    P(D′) := D
               end if
          end if
end for
O := O - {D}
C := C + {D}
end proc
```

где O – список «открытых», C – список «закрытых» узлов, $g(D)$ – минимальная известная длина пути от D_0 до D, $P(D)$ – предшествующий узел на найденном пути. В этом алгоритме поиска на графе есть две функции – `Select` и `Open`. Неформально говоря, первая из них есть «поиск», а вторая – «на графе».

3.2. Эвристический поиск

Функция `Select` каким-то образом определяет очередной узел для раскрытия.

Если `Select (O) = FIFO` – происходит поиск в ширину; этот метод гарантирует нахождение пути, если он существует.

Если `Select (O) = LIFO` – происходит поиск в глубину; этот метод подобен поиску с возвратами, но хранит уже не один текущий, а все просмотренные пути.

Это все неинформированные процедуры поиска. Без использования дополнительной информации о задаче оба метода безнадежно плохи.

3.2.1. Эвристики

Конечно, выбор очередного узла для раскрытия делается более тонким образом, с применением эвристической информации.

Метод поиска, использующий эвристическую информацию, называется *эвристическим поиском*.

Эвристики уменьшают вычислительные затраты на применение правил за счет того, что сокращают общее количество применений правил. Но применение самих эвристик также требует каких-то вычислений и привносит свои затраты. Кроме того, манипуляции с графом поиска, особенно если он велик, также требуют затрат. На самом деле, нас интересует общая сумма, складывающаяся из затрат на применение правил, затрат на проверку и раскрытие узлов и затрат на эвристику. Точнее говоря – мы хотим минимизировать среднее значение этой суммы по всем задачам.

3.2.2. Оценочная функция

Эвристическая информация может иметь любую форму. В ИИ принято представлять эвристическую информацию в виде *оценочной функции* $F(D)$.
Обычно оценочная функция F определена на множестве допустимых состояний базы данных и принимает вещественные числовые значения $F : D \to \mathbf{R}$. Причем обычно выбирают оценочную функцию таким образом, чтобы на терминальных узлах ее значение было минимально $F(D_t) = \min F(D)$. Если есть оценочная функция, то можно записать функцию `Select`.

```
Proc Select (O)
m := ∞
D' := nil
for D ∈ O do
     if F(D) < m then
          D' := D
          m := F(D)
     end if
end for
return (D')
end proc
```

Или, короче:

```
Select(O) := min F(D), D ∈ O
```

Для оценочной функции F есть много различных идей о ее воплощении. Слово «оценочная» означает, что F (предположительно) близка к каким-либо важным характеристикам поиска. Ниже приведен список некоторых характерных свойств, которые часто пытаются использовать при выборе оценочной функции.

- Величина, обратная вероятности того, что D принадлежит оптимальному пути.
- Расстояние от D до целевого множества узлов.
- Стоимость пути от исходной вершины D_0 до целевой вершины D_t через текущую вершину D.

Мы рассматриваем последний случай.
Пример. Игра в 8.

Введем оценочную функцию F как сумму глубины в дереве и числа шашек не на месте (рис. 22).

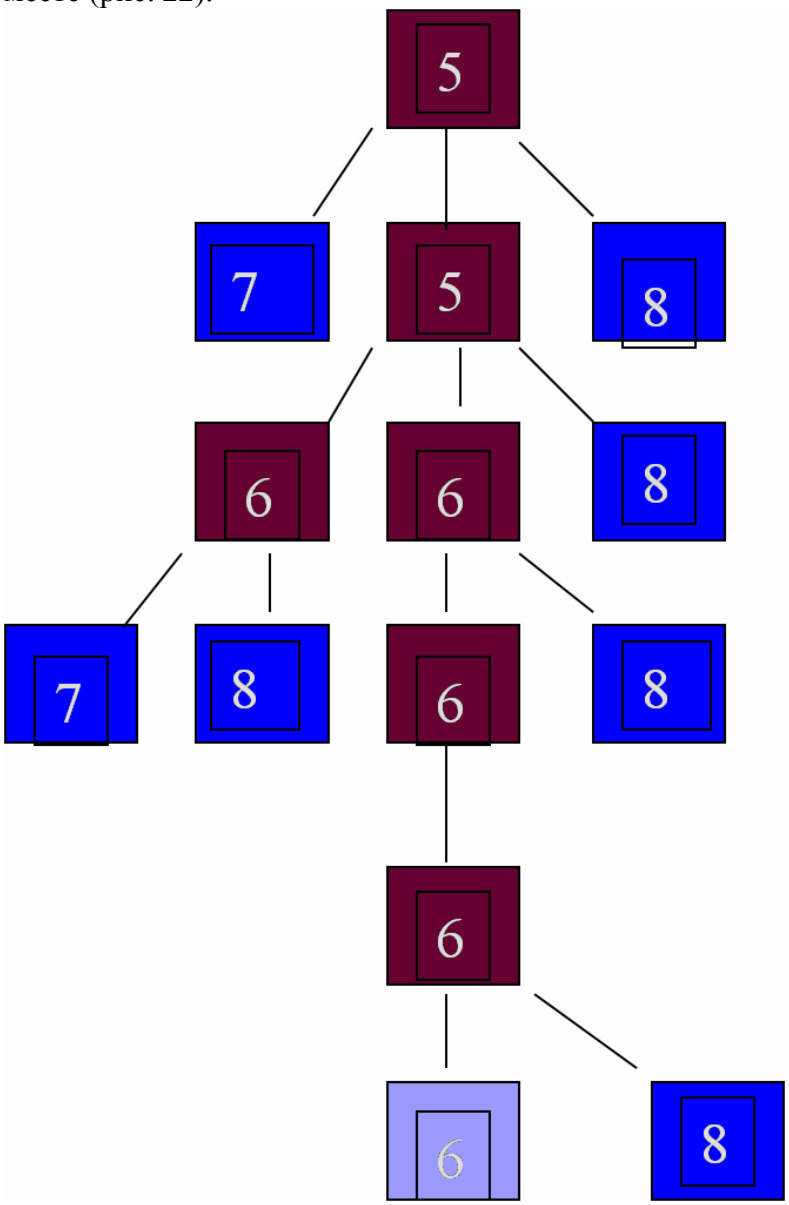


Рис. 22. Дерево игры в 8 со значением оценочной функции F.

Если взять F просто как глубину в дереве, то получится поиск в ширину.

3.3. Свойства алгоритма GS

Пусть $F^*(D) = g^*(D) + h^*(D)$ – точная длина кратчайшего пути от D_0 *к* D_t через D, где

g^* – длина кратчайшего пути от D_0 *к* D,

h^* – длина кратчайшего пути от D *к* D_t:

$g^*(D) = \min d(D_0, D)$, $h^*(D) = \min d(D, Dt)$.

Введем обозначение:

$M := F^*(D_0) = h^*(D_0)$ – это оптимум, длина кратчайшего пути.

Заметим, что функции F^*, g^*, h^* и число M нам, вообще говоря, неизвестны.

Рассмотрим оценочную функцию в следующей форме
$F(D) = g(D) + h(D)$, где g – уже определена нами в `Open`.
Это оценка: $g(D) \geq g^*(D)$, так как возможно не все пути еще построены.
Под вопросом остается функция $h(D)$ – оценка расстояния от текущего положения до решения.

Теорема.
Если $\forall D\ h(D) \leq h^*(D)$*, то GS состоятелен* (*частично корректен*).

Говорят, что алгоритм поиска *состоятелен* (частично корректен), если он находит оптимальный путь, при условии что такой путь существует.

Доказательство теоремы основано на серии лемм. Все леммы доказываются при условии, что $h \leq h^*$.

Лемма 1
Если алгоритм GS не заканчивает свою работу, то оценка раскрываемого узла неограниченно возрастает. То есть $\forall N\ \exists k\ F(D_k) > N$.
Доказательство.
Пусть $e = \min r.c > 0$, p – глубина узла D в дереве. Тогда $p > \log_{|R|} k$.
Возьмем $k > |R|^{N/e}$. Имеем: $F(D_k) > g(D_k) > e\ p > e \log_{|R|} k > e \log_{|R|} |R|^{N/e} = N$. **ч.т.д.**
Лемма 2
Если существует оптимальный путь, то в списке открытых узлов всегда есть узел, принадлежащий этому пути, оценка которого не превосходит M.
Доказательство.
Пусть $A = \langle D_0, D_t \rangle$ – оптимальный путь. По индукции. База: $D_0 \in A$. Открытие узла из A оставляет наследника в O. Значит узел есть. Пусть теперь $D' \in O$, $D' \in A$, и D' – самая левая, т.е. $g(D') = g^*(D')$. Тогда

$$F(D) = g(D')+h(D') = g^*(D')+h(D') \leq g^*(D')+h^*(D') = F^*(D') = M,$$

т.к. $\forall D \in A\ F^*(D) = M$. **ч.т.д.**
Лемма 3
Если существует оптимальный путь, то открывается всегда узел, оценка которого не превосходит M.
Доказательство.
По лемме 2 $\exists D \in O\ F(D) \leq M$, но **`Select`** выбирает минимум **ч.т.д**.
Доказательство теоремы

- Пусть $\exists A$ и алгоритм не заканчивает свою работу. Тогда по лемме 1 $\exists k\ F(D_k) > M$, что противоречит лемме 3. Значит заканчивает.
- Алгоритм *GS* может закончить **fail** и **OK**. **Fail** если O пуст. Но по лемме 2 $O \neq \varnothing$, значит **ОК**.
- Пусть *GS* закончил **ОК**, но P – не оптимальный путь. $h^*(D_t)=0 \rightarrow h(D_t)=0$. $F(D_t) = g(D_t)+h(D_t) = g(D_t) > g^*(Dt) = F^*(D_t) = M$, что противоречит лемме 3, **ч.т.д.**

Следствия
1. *Любая вершина* $D \in O$*, для которой* $F(D) < M$*, будет раскрыта*.
Доказательство. Последним выбирается для раскрытия узел D_t $F(D_t)=M$, а раскрываются с минимальной оценкой. **Ч.т.д.**

2. *Поиск в ширину состоятелен*.

Доказательство. Поиск в ширину: $h = 0$, $F = g$, $0 \le h^*$, **ч.т.д.**

Замечание.

Рассмотренный алгоритм известен в литературе также под названием A* (читается «А со звездочкой»).

3.4. Сравнение оценочных функций

Пусть даны две оценочных функции,

$F_1(D) = g_1(D) + h_1(D)$,

$F_2(D) = g_2(D) + h_2(D)$,

Причем выполнены условия теоремы о состоятельности

$h_1(D) \le h^*(D)$

$h_2(D) \le h^*(D)$

Пусть $h_2(D) > h_1(D)$ – более информированная функция. Тогда в случае применения F_2 будет раскрыто не более узлов, чем в случае применения F_1.

Теорема.

Если существует оптимальный путь, то к моменту окончания поиска всякий узел, раскрытый по F_2, раскрыт и по F_1.

То есть более информированная оценочная функция быстрее находит решение.

Доказательство (рис. 23).

Индукция по k – глубине узла D в дереве поиска по F_2.

База. D_0 либо оба раскрывают, либо оба не раскрывают ($D_0 = D_t$).

Пусть D – узел выбранный для раскрытия по F_2 на шаге k. По индукционному предположению до D по F_1 было раскрыто не меньше, чем по F_2, значит $g_1(D) \le g_2(D)$. Далее от противного. Пуст алгоритм закончен и D не раскрыт по F_1. По следствию 1 к теореме о состоятельности $F_1(D) \ge M$. Но по F_2 узел D раскрыт, значит $F_2(D) \le M$. Имеем $g1(D) + h1(D) \ge g2(D) + h2(D) \ge g1(D) + h2(D)$, откуда $h1(D) \ge h2(D)$, что противоречит условию. **Ч.т.д.**

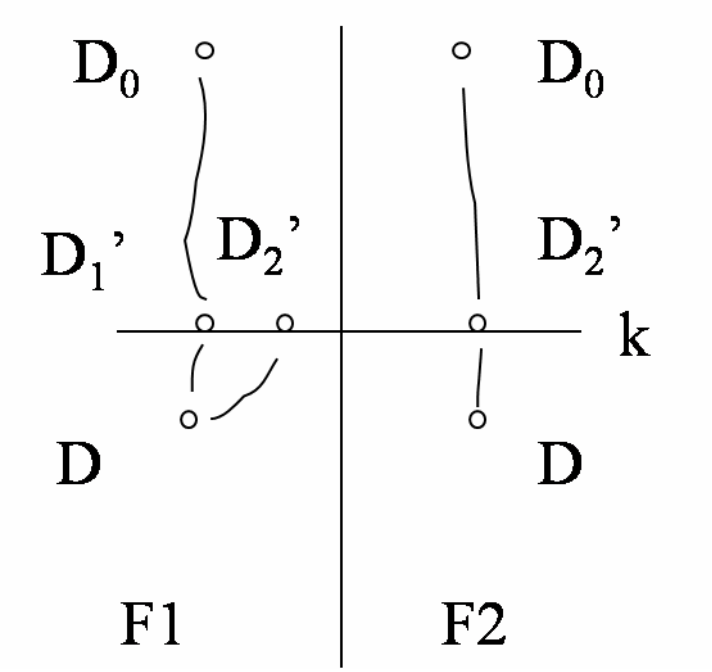


Рис. 23. К Доказательству теоремы о сравнении оценочных функций.

3.5. Монотонное ограничение

Говорят, что функция h удовлетворяет *монотонному ограничению*, если

$$\forall D\ h(D) \le h(r.f(D))+r.c\ \&\ h(D_t)=0.$$ (Неравенство треугольника)

Другими словами эвристическая функция локально согласована со стоимостями дуг (правил).

Теорема.

Если выполнено условие состоятельности, оценочная функция удовлетворяет монотонному ограничению и существует оптимальный путь, то GS всегда выбирает для раскрытия вершину, к которой уже найден оптимальный путь, т.е.

$$g(D) = g^*(D).$$

Доказательство.

Метод математической индукции по длине оптимального пути.

База: $g(D_0) = g^*(D_0) = 0$.

Пусть выбрана для раскрытия вершина $D \neq D_0$ и пусть $A=D_0, D_1, \ldots, D_i, \ldots, D$ – оптимальный путь.

$\forall i\ g^*(D_i)+h(D_i) \le g^*(D_i)+c(D_i,D_{i+1})+h(D_{i+1})$ по монотонному ограничению. Далее

$D_i, D_{i+1} \in A \rightarrow g^*(D_i)+c(D_i,D_{i+1}) = g^*(D_{i+1})$. Имеем

$\forall i\ g^*(D_i) + h(D_i) \le g^*(D_{i+1}) + h(D_{i+1})$ и по транзитивности

$\forall i\ g^*(D_i) + h(D_i) \le g^*(D) + h(D)$.

Пусть теперь $D_k \in A,\ D_k \in O\ \&\ \forall j<k\ D_j \in C$. Такой узел существует, т.к.

$D_0 \in C,\ D \in O$. Все вершины на A левее D_k уже раскрыты, значит $g(D_k) = g^*(D_k)$.

$$g(D)+h(D) = F(D) \le F(D_k) = g(D_k)+h(D_k) = g^*(D_k)+h(D_k) \le g^*(D)+h(D)$$

Следовательно, $g(D) \le g^*(D)$. Но $\forall D\ g(D) \ge g^*(D)$ и значит $g(D) = g^*(D)$. **ч.т.д.**

Следствие.
Если оценочная функция удовлетворяет монотонному ограничению, то последовательность значений оценочной функции для выбираемых узлов не убывает.
Доказательство.
Пусть раскрыты узлы D_1, D_2 (подряд).
Если в момент раскрытия D_1 уже $D_2 \in O$, то по построению GS имеем $F(D_1) \le F(D_2)$.
Иначе D_2 должен попадать в O при раскрытии D_1.
$F(D_2) = g(D_2) + h(D_2) = g^*(D_2) + h(D_2) = g^*(D_1) + c(D_1, D_2) + h(D_2) =$
$= g(D_1) + c(D_1,D_2) + h(D_2) \ge g(D_1) + h(D_1) = F(D_1)$ **ч.т.д.**

3.6. Поиск на графах И/ИЛИ

При рассмотрении разложимых систем продукции были введены графы И/ИЛИ. В этом графе вершины – это состояния базы данных или их части, которые получаются двумя способами: либо применением правила, либо разложением базы данных на составляющие. Это именно граф, а необязательно только дерево поиска. Одна и та же вершина D может появиться и в результате разложения D_1 и применением правила к D_2.

3.6.1. Граф И/ИЛИ

Уточним определения.

Граф И/ИЛИ – это гиперграф. *Гиперграф* – это пара, состоящая из множества вершин $\{D\}$ и множества k-дуг. k-дуга – это пара $D \to (D_1,...,D_k)$, причем 1-дуга – это просто дуга.

Так же как и в обычном орграфе, в гиперграфе можно определить понятие пути (гиперпути).

Гиперпуть в гиперграфе – это последовательность множеств узлов, $<S_1, \ldots, S_n>$, причем для каждого узла $D \in S_i$, $1 \le i \le n-1$, существует ровно одна k-дуга $D \to (D_1,...,D_k)$, такая, что $\forall 1 \le j \le k\ (D_j \in S_{i+1})$.

Определение гиперпути такое же, как определение пути, но гиперпуть в гиперграфе будет выглядеть как граф, а не как одна чередующаяся последовательность узлов и дуг. Из каждой вершины выбирается ровно одна k-дуга. Из каждой полученной вершины снова выбирается ровно одна k-дуга т.д.
Среди всевозможных гиперпутей, нам интересны те, которые начинаются в исходном состоянии базы данных и заканчиваются в терминальных состояниях.

Граф решения $G(D_0,T)$ – это гиперпуть из D_0 в множество терминальных узлов T.

Можно эквивалентное рекурсивное определение графа решения $G(D_0, T)$:
Если $D_0 \in T$, то $G = D_0$.
Если $D_0 \notin T$, и из D_0 исходит k-дуга $D \to (D_1,\ldots,D_k)$, причем существуют графы решения $G_1, G_2, \ldots, G_k$ для узлов $D_1, D_2, \ldots, D_k$, то $G = D_0 \to (G_1,\ldots,G_k)$.

Стоимость графа решения (то есть длина пути в гиперграфе) определяется с помощью индуктивного определения:

$$c(G(D,T)) := c(D \rightarrow (D_1,\ldots,D_k)) + \Sigma\, c(G(D_i,T))$$

Таким образом, одна и та же k-дуга при подсчете длины может учитываться много раз.

Пример (рис. 24). Считая, что узлы 6 и 7 терминальные, а узел 0 начальный, и принимая допущение, что стоимость k-дуги равна k, получаем что в этом графе И/ИЛИ имеются два гиперпути из 0

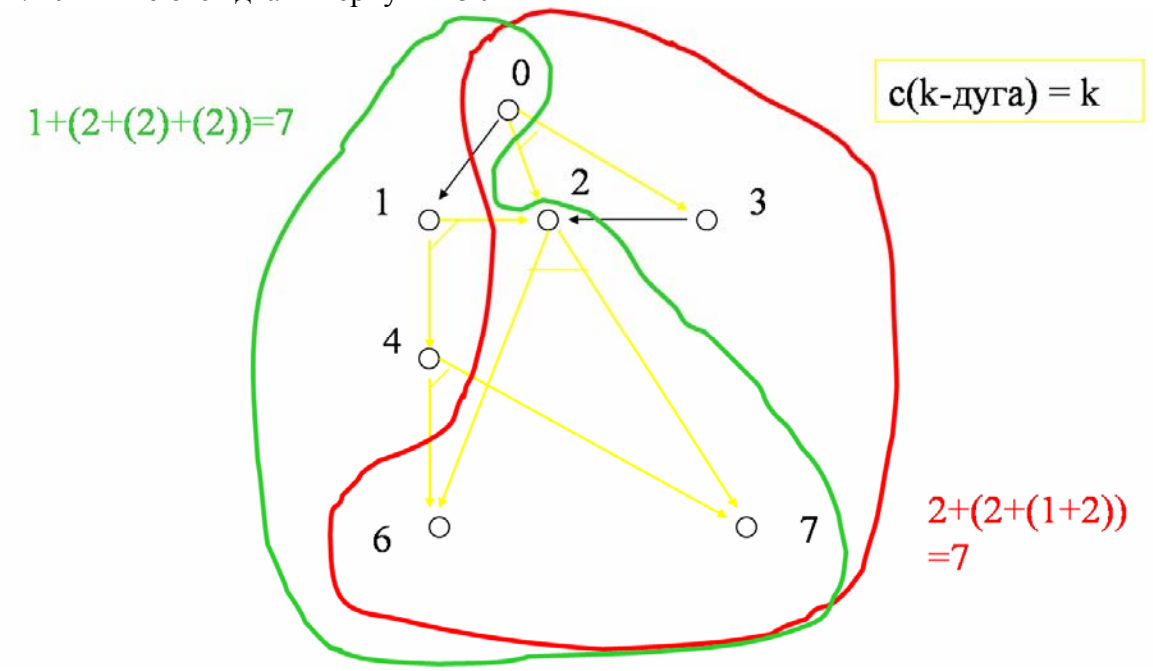


Рис. 24. **Гиперграф решения и *k*-дуги**.

Если считать что длина k-дуги равна k, то на рис. 24 можно выделить два графа решения (на рисунке они обведены).

Длина первого графа решения = 1+(2+(2)+(2)) = 7,

Длина второго графа решения = 2+(2+(1+2)) = 7,

так как согласно индуктивному определению некоторые k-дуги могут учитываться несколько раз.

3.6.2. Поиск на графе И/ИЛИ

Алгоритм эвристического поиска на графе И/ИЛИ заключается в следующем.

Пусть наш граф ациклический, имеется оценочная функция F, и h удовлетворяет условию монотонности (монотонному ограничению):

$$h(D) \le c(D,(D_1,\ldots,D_k)) + \Sigma\, h(D_i)$$

и условию состоятельности (то есть является нижней оценкой):

$$h(D) \le h^*(D).$$

Процедуры поиска на графах И/ИЛИ определяются следующим образом.

```
proc GS и/или (D0, R, T)
     G := D0    {частичный граф решения}
     s(D0) := D0 ∈ T {отметка узла}
     while ¬s(D0) do
```

```
          D := Select (G)
          Open (D,G)
     end while
end proc
```

где s(D): 0..1 – отметка узла, означающая, найден ли для этого узла гиперпуть, ведущий в терминальные вершины.

Процедура Select на основе построенного графа поиска рассматривает частичный граф решения и выбирает для раскрытия на этом графе нераскрытый узел минимальной стоимости.

```
proc Select (G)
     O := {D0}
     for D∈ O do
          for q = (D→D1,…,Dk) ∈  Q(D) do
               if x(D→D1,…,Dk) then
                    O:=O - {D}
                    {D1,…,Dk} := {D1,…,Dk} - T
                    O:=O + {D1,…,Dk}
               end if
          end for
     end for
     m := ∝
     for D ∈ O do
          if F(D)<m then
               D' := D
               m := F(D)
          end if
     end for
     return (D')
end proc
```

В этой процедуре фактически каждый раз заново прогоняется фронт решения от D_0 к концевым вершинам по отмеченным *k*-дугам.

Здесь используются следующие обозначения.

O – концевые вершины частичного графа решения, Q(D) – множество дуг узла D, x(q) – отметка дуги q.

Процедура Open значительно сложнее. Она раскрывает выбранную вершину, пересчитывает оценки вершин, отмечает *k*-дуги и отмечает s (solved) вершины. Эти изменения могут затронуть весь граф.

В этой процедуре мы считаем, что понятие правила обобщает как применение правила, так и расщепление базы данных с помощью *k*-дуги.

```
proc Open (D,G){раскрытие вершины}
```

```
     for r∈ R do
          if r.p(D) then
               G := G + (D→ D1,…,Dk)
               for i from 1 to k do
                    if Di∉ G then F(Di):=h(Di) end if
                    s(Di) := Di ∈ T
               end for
          end if
     end for
     { Пересчет оценок и отметок }
     C := {D}
     while C ≠ ∅ do
          {выбор вершин для пересчета}
          for D'∈ C do
               if P(D') ∩ C = ∅ then D:=D' end if
          end for
          C := C - {D}
          m := F(D)
          for q=(D→ D1,…,Dk) ∈ Q(D) do
               c':=c(q) + Σ F(Di)
               if c' < m then
                    x(q(D)) := 0
                     x(q) := 1
                    s(D):=&s(Di)
                    m := c'
               end if
          end for
          if s(D) ∨ F(D)≠m then
               F(D):=m
               C := C + U(D)
          end if
     end while
end proc
```

Здесь используются следующие обозначения.

Q(D) – множество исходящих из D дуг (в G),

P(D) – множество потомков D в G,

q(D) – отмеченная дуга из D,

`x(q)` – отметка дуги,
`U(D)` – множество предков `D` в `G` вдоль отмеченных дуг,
`s(D)` – отметка узла, означающая что из него построен гиперпуть до цели.

Процедура `Select` строит фронт текущего частичного графа решения по отмеченным дугам и выбирает узел с наименьшей оценкой в этом фронте.
Процедура `Open` сначала раскрывает выбранную вершину, а затем пересчитывает оценки узлов, отмечает дуги и вершины. При этом данная процедура может переоценить весь граф.

3.6.3. Пример применения процедуры поиска на графе И/ИЛИ

На рис. 25–рис.29 приведен пример работы алгоритма поиска на графе И/ИЛИ применительно к графу, представленному на рис. 24.
На этих рисунках используются следующие обозначения. Состояния базы, то есть узлы графа И/ИЛИ обозначаются кружками. Если кружок закрашен, то это означает, что для него получено условие `s(D) = 1`, то есть найден граф решения из данного узла в терминальные узлы (которыми в данном примере являются узлы 5 и 6). 1-дуги обозначаются черным цветом, *k*-дуги (в этом примере *k*=2) обозначаются более светлым цветом и имеют небольшую черточку, соединяющую ветви *k*-дуги. Число рядом с узлом является текущей оценкой этого узла. При это считается, что функция *h* для терминальных узлов имеет значение 0, а для нетерминальных – 1. Длина *k*-дуги равна *k*. Отмеченная дуга, исходящая из данного узла, отмечена жирной полоской.

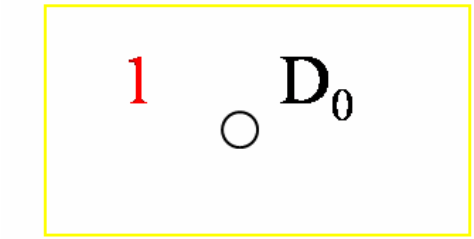


Рис. 25. Пояснение к процедурам поиска на графе И/ИЛИ. Шаг 1. Начальное состояние.

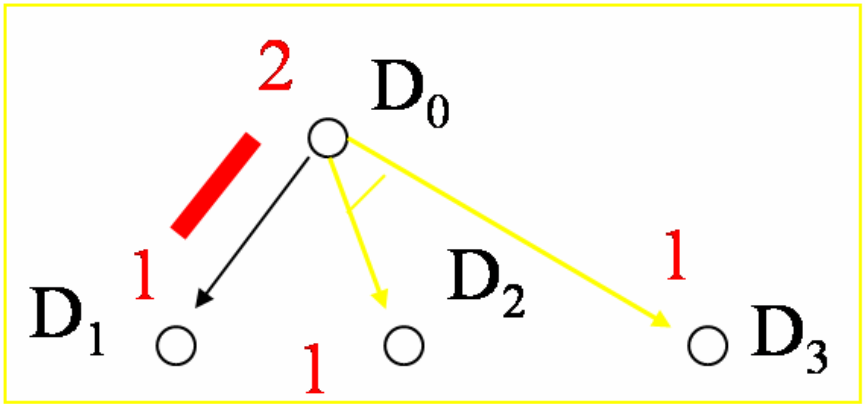


Рис. 26. Пояснение к процедурам поиска на графе И/ИЛИ. Шаг 2. Первое раскрытие узла.

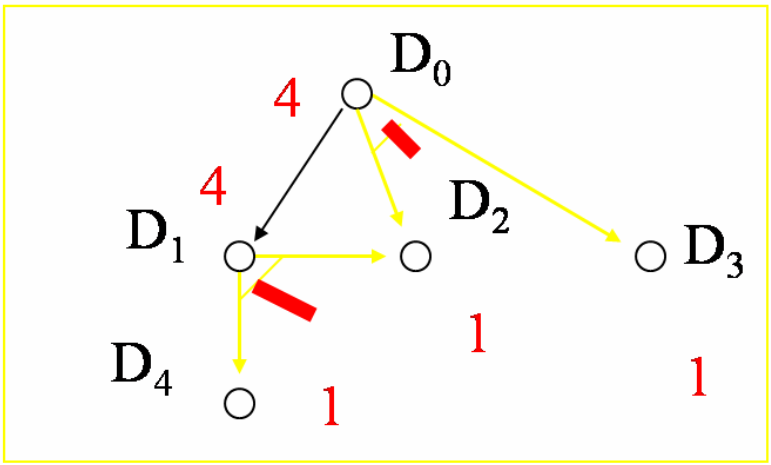


Рис. 27. Пояснение к процедурам поиска на графе И/ИЛИ. Шаг 3. Второе раскрытие узла.

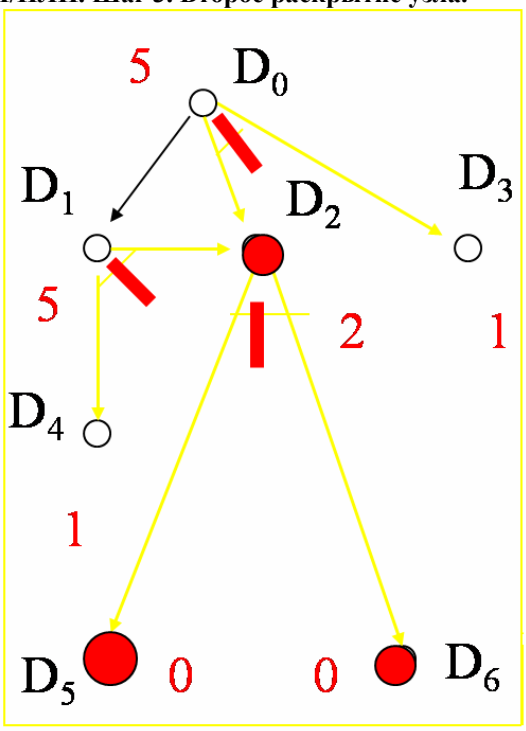


Рис. 28. Пояснение к процедурам поиска на графе И/ИЛИ. Шаг 4. Третье раскрытие узла.

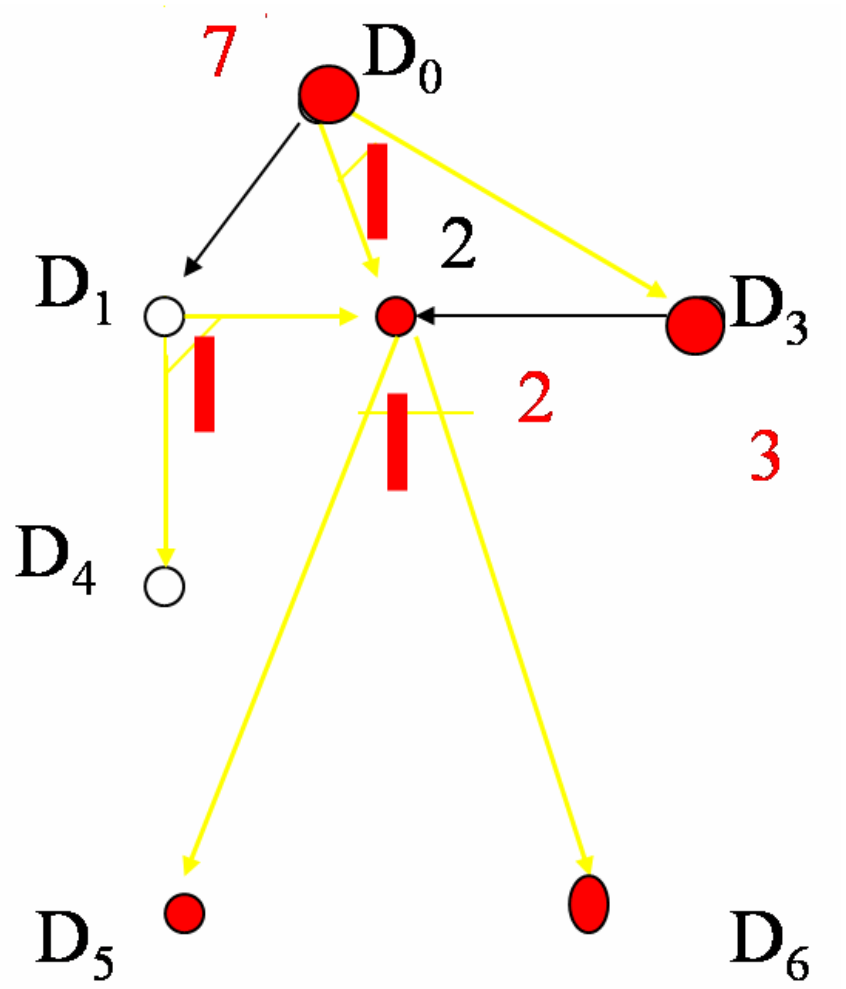


Рис. 29. Пояснение к процедурам поиска на графе И/ИЛИ. Шаг 5. Построение графа решения.

3.7. Поиск на игровых деревьях

Допустим, имеется антагонистическая дискретная игра с полной информацией. В ней принимают участие два игрока Max и Min. Пусть первым ходит Max.

Составим дерево игры – граф И/ИЛИ. Примем 1-дуги из узлов для первого игрока Max (И), k-дуги из узлов для второго игрока Min (ИЛИ). Попробуем составить выигрышную стратегию для игрока Max.

Для простоты будем считать граф игры деревом, размножив совпадающие позиции.

Правилами игры задана оценка F^* конечной позиции: $F^* = +\infty$ если победил Max, $F^* = -\infty$ – победил Min и $F^* = 0$ если закончилась вничью.

3.7.1. Минимакс

Минимаксный метод решения заключается в следующем. Пусть есть некоторая статическая оценочная функция F, причем если позиция выгодна для Max, то $F > 0$, если позиция выгодна для Min, то $F < 0$. Если бы Max выбирал среди концевых

вершин, то он бы выбирал наибольшее значение *F*. Если бы Min выбирал среди концевых вершин, то он выбирал бы наименьшее значение *F*.

Если бы у нас было полное дерево игры, мы бы начали снизу, с терминальных вершин, оценка которых задана правилами игры, и все было бы достаточно просто – игра была бы решена.

Но полное дерево слишком велико. К примеру, для шашек это 10^{40} позиций, а для шахмат – 10^{120}. Но можно эту идею (начать снизу) применить и к частичному дереву.

Для этого строим частичное дерево, статической функцией оцениваем узлы и возвращаемся по дереву назад.

Пример. Рассмотрим игру в крестики-нолики на доске 3×3.

Применение метода Минимакс (Minimax) с глубиной частичного дерева игры 2 приведен на рисунке 30 («крестики» обозначены цифрами 1, а «нолики» – цифрой 0). Число после знака = в позиции – это ее оценка.

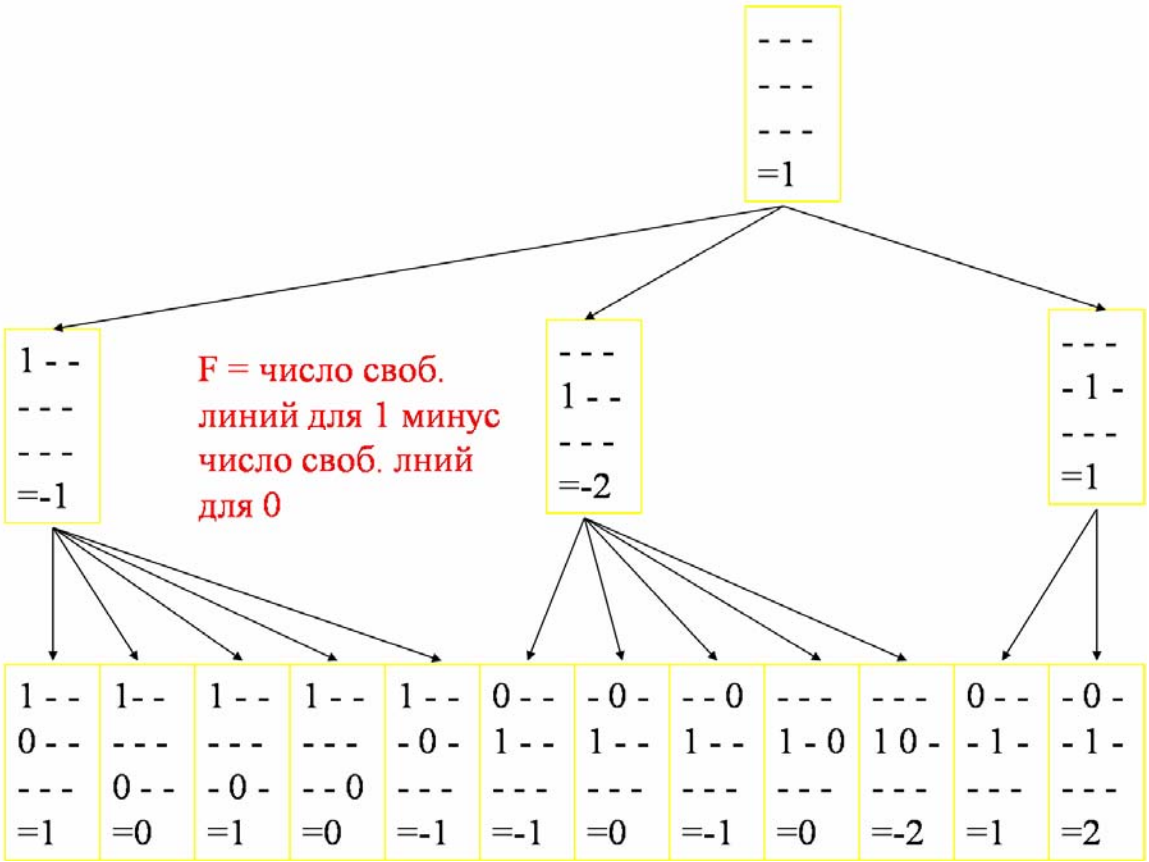


Рис. 30. Дерево игры Минимакс.

Мы видим, что наилучшим ходом для Max при оценке по методу Минимакс с глубиной 2 оказывается ход в центральную клетку. Как нетрудно сообразить, это действительно выигрышная стратегия для Max.

3.7.2. α-β отсечение

В Минимаксе порождение позиций и их оценка отделены друг от друга. Сначала происходит порождение, а только потом это порождение оценивается. Можно

производить и одновременную оценку прямо во время порождений, чтобы не порождать ничего лишнего. Такая процедура называется α-β *отсечение*.
Рассмотрим ряд достаточно логичных соображений, на которых основано α-β отсечение.
Пусть α – граница снизу для значения оценки в данном узле, β – граница сверху для значения оценки в данном узле.
Заметим, что значение α для Max узла не могут уменьшаться, а значение β для Min не могут увеличиваться. Следовательно:
1. Можно не искать из узла Min, если $\beta \le \alpha$ для всех его Max родителей.
2. Можно не искать из узла Max, если $\alpha \ge \beta$ для всех его Min родителей.

Пример для игры в крестики-нолики приведен на рисунке 31.

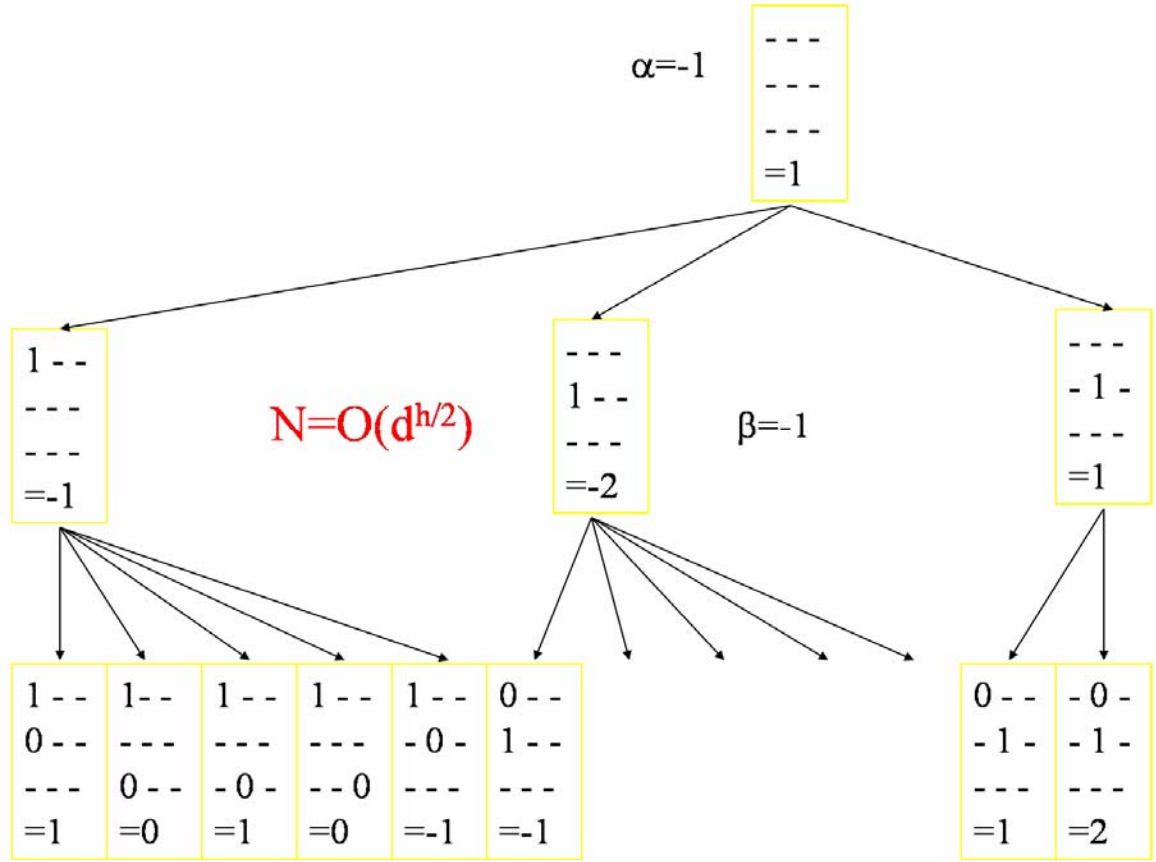


Рис. 31. α-β отсечение

Мы видим, что четыре узла на третьем уровне можно не порождать их порождение не изменит оценки остальных узлов.

3.7.3. Эффективность α-β отсечения

Насколько эффективно α-β отсечение?
Пусть глубина дерева h, а каждая вершина имеет ровно d сыновей.
Тогда внизу имеется ровно $N=d^h$ узлов.
Пусть (наилучший случай) вершины для Min порождаются в порядке возрастания, а для Max – в порядке убывания значений оценочной функции.

Тогда можно показать, что при использовании α-β отсечения число узлов будет равняться:

$$N=O(d^{h/2}) = \begin{cases} 2d^{h/2}, \text{ если h четно} \\ d^{(h+1)/2} + d^{(h-1)/2} - 1, \text{ если } h \text{ нечетно.} \end{cases}$$

Таким образом, в идеальном случае α-β процедура позволяет удвоить глубину поиска по сравнению с Минимаксом и в любом случае не хуже Минимакса.

Практика показывает, что сила игры программы прямо пропорциональна глубине поиска. Например, начинающий шахматист редко просматривает позицию больше чем на три-четыре полухода вперед. Т.е. глубина дерева поиска у новичка примерно 3. Шахматные мастера строят в уме дерево на дюжину полуходов, а если намечается форсированный вариант, то и больше. Гроссмейстеры и чемпионы видят еще дальше, на два десятка полуходов. Находясь в миттельшпиле, они уже могут предсказать эндшпиль.

Каким же образом люди не проигрывают компьютерам в шахматы? Кстати, не проигрывают уже только гроссмейстеры и чемпионы. Средний любитель шахмат проиграет современной шахматной программе с вероятностью 100%. Дело в том, что шахматист человек строит в уме гораздо более узкое дерево. В сложной шахматной позиции шахматист сразу видит 2-3 наилучших хода из двух десятков возможных ходов и только их и просчитывает в глубину. А шахматная программа, даже применяя наилучшие известные приемы, подобные α-β отсечению, отбрасывает в лучшем случае только половину возможных ходов. Если у человека ширина ветвления равна 2, а у программы 10, то для поиска на глубину 10 человеку нужно просмотреть 2^{11} = 2048 позиций, а программе 10^{11} = 1 000 000 000 000 позиций.

Тема 4. Представление знаний формулами исчисления предикатов

Материал этой темы базируется в основном на математической логике, основные понятия которой считаются известными. Естественно, применение логических исчислений в ИИ рассматривается на конкретных примерах.

4.1. Язык исчисления предикатов первого порядка

Основным инструментом ПЗ, рассматриваемым в этой главе, являются формулы исчисления предикатов первого порядка.

4.1.1. Грамматическое описание

Ниже приведена формальная грамматика языка исчисления предикатов первого порядка.

Терм :	Конст | Перем | Функ (Список термов)
Список термов :	Терм | Терм , Список термов
Атом :	Предикат | Предикат (Список термов)
Литерал :	Атом | ¬ Атом
Формула :	Атом | ¬ (Формула) |
	(Формула Связка Формула) |
	Квантор Перем (Формула)

Связка : $\wedge$ | $\vee$ | $\rightarrow$
Квантор : $\forall$ | $\exists$

Мы рассматриваем язык, в котором участвуют логические связки, кванторы и атомарные формулы, построенные над множеством функциональных символов, предметных переменных и констант. Понятие *литерала* (атом или отрицание атома) вообще говоря, грамматически излишне. Мы ввели его, чтобы иметь под рукой этот термин.
Необходимо сделать несколько замечаний, касающихся грамматики этого языка.

- Во-первых, в этом языке мы считаем, что наши функциональные и предикатные символы имеют фиксированное число аргументов, как в обычных системах программирования. Это довольно сильное допущение, и можно, в принципе, так не считать. В этом случае мы будем иметь дело с λ-выражениями и смешанными вычислениями, что выходит за рамки данного пособия.
- Во-вторых, язык, который мы используем, в логическом программировании рассматривается как язык первого порядка. Другими словами это означает, что под кванторами могут стоять только предметные переменные, т.е. не функциональные символы и не предикаты. В противном случае это будет уже исчисление не первого порядка. В реальных современных системах ИИ используются именно такие исчисления. Например, модное направление Model Checking основано на темпоральных логиках, которые выглядят как логики первого порядка, а на самом деле сводятся к стандартным логикам не первого порядка.

- В-третьих, грамматика математической логики предполагают формальную запись утверждений в определенном синтаксисе. В нашем рассмотрении мы, естественно, сильно упростим формальный синтаксис и будем записывать формулы, как удобнее, однако так, чтобы всегда можно было догадаться, какие синтаксические упрощения используются в том или ином случае. В частности, используется обычный приоритет связок и кванторов, лишние скобки опускаются.
- В-четвертых, известно, что чистое исчисление предикатов первого порядка имеет ограниченную выразительную силу. Другими словами, если не делать внелогических допущений, то многие содержательные теории невыразимыми или трудно выразимыми в чистом исчислении первого порядка. Чистого исчисления предикатов первого порядка для этого недостаточно. Естественно, что мы столкнемся с внелогическими встроенными константами, функциональными символами и предикатами. В частности, считается, что натуральные числа и другие общеизвестные понятия можно использовать в любых формулах, не пытаясь аксиоматически описать, что на самом деле это такое.
- В-пятых, замкнутых формул вполне достаточно, и можно не рассматривать незамкнутые формулы, потому что если есть незамкнутая формула, то она будет выполнима тогда и только тогда, когда выполнимо соответствующее замыкание – это эквивалентно утверждению, что все свободные переменные замыкаются квантором всеобщности.

4.1.2. Интерпретация

Интерпретация – это множество функций I, которое отображает формальную теорию T в конкретную предметную область M:

$$I : T \to M$$

Множество функций нужно, потому что требуется каждой формальной константе сопоставить фактический объект, формальным функциональным символам сопоставить конкретные операции, формальным предикатам сопоставить конкретные отношения. После этого формальное утверждение становится содержательным, и с содержательной точки зрения мы можем проверить, истинно оно или ложно.

Истинностное значение формулы A в интепретации I обозначается $I(A)$.

Смысл формальной теории (если она адекватна) состоит в том, что истинными оказываются такие утверждения, которые одновременно обязательно формально доказуемы.

Представления знаний с помощью исчисления предикатов базируется на следующей идее. Мы будем строить формальные теории, проводить с их помощью поиск формальных доказательств и интерпретировать полученные результаты в предметной области, где работает искусственная интеллектуальная программа.

При этом мы часто будет пользоваться следующими понятиями:

Логическое следование. Говорят, что формула F логически следует из формулы A, если формула F истинна во всякой интерпретации, в которой истинна формула A

$$A \Rightarrow F := I(A) \to I(F)$$

Формальные теории имеют некоторое множество правил вывода, которые обозначаются *R*.

Выводимость. Говорят что формула *F* формально выводима из формулы *A*, если существует последовательность формул <*A*,…,*F*>, в которой каждая формула (кроме первой) выводима из некоторых предшествующих формул по одному из правил вывода *R*

$$A \vdash_R F := \exists \langle A,\ldots,F\rangle$$

Логическое следование – это семантическое понятие. Выводимость – это синтаксическое понятие. Они могут быть, а могут и не быть связаны. Все зависит от правил вывода *R*.

Правила *R* *логичны*, если из формальной выводимости получается логическое следование

$$A \vdash_R F \to A \Rightarrow F$$

Правила *R* *полны*, если любое содержательное логическое следование подтверждено формальной выводимостью

$$A \Rightarrow F \to A \vdash_R F.$$

4.2. Метод резолюций

В 1960 году Робинсон предложил метод резолюций. Этот метод применим в логических исчислениях, в которых используются формулы особого вида, называемые предложениями (clause).

4.2.1. Правило резолюции

Предложение – это бесквантерная дизъюнкция литералов.

Любую формулу исчисления предикатов первого порядка можно свести к бесквантерной дизъюнкции литералов. Это сведение не является логически эквивалентным, но, тем не менее, оказывается верным следующее утверждение:
те формулы, которые получились в результате сведения исходных формул к предложениям, образуют противоречивое множество тогда и только тогда, когда исходное множество формул не выполняется.
На основе этого механизма используется исчисление, именующееся *метод резолюций*, в котором есть только одно правило вывода – *правило резолюции*.

Джон Алан Робинсон В 1963 г. предложил метод автоматического доказательства теорем, получивший название "метод резолюций". Идея этого метода принадлежит Эрбрану, но поскольку тот не думал о применении своего метода на компьютерах (что не удивительно в 1930 г.), он оказался не очень подходящим с вычислительной точки зрения. Робинсон сделал его пригодным для реального применения и разработал эффективный алгоритм унификации, являющийся основой метода.	

Если даны два предложения (две бесквантерные дизъюнкции литералов), причем в них есть контрарные литералы, (P и ¬P), то есть из этой пары предложений можно вывести новое предложение, выбрасывая контрарные литералы.
Как обычно, в теории логического вывода правила вывода записывают следующим образом: над чертой посылки, под чертой заключение.

$$\frac{P \vee P_1 \vee \mathrm{K} \vee P_n \qquad \neg P \vee Q_1 \vee \mathrm{K} \vee Q_m}{P_1 \vee \mathrm{K} \vee P_n \vee Q_1 \vee \mathrm{K} \vee Q_m}$$

Таким образом, правило резолюции применимо только к предложениям, содержащим контрарные унифицируемые литералы. Заключение (*резольвента*) – это объединение всех литералов за исключением пары контрарных.
Многие правила (например, Modus ponens и транзитивность) – это все частные проявления правила резолюции.
Ниже в таблице слева приведены названия, в центре – традиционные формы известных правил вывода, справа – те же самые правила, записанные в ограничениях метода резолюции в виде бесквантерной дизъюнкции литералов.

Название правила	Традиционная форма	Резолютивная форма
Modus ponens	$\frac{A, \quad A \to B}{B}$	$\frac{A, \quad \neg A \vee B}{B}$
Транзитивность	$\frac{A \to B, \quad B \to C}{A \to C}$	$\frac{\neg A \vee B, \quad \neg B \vee C}{\neg A \vee C}$
Слияние	$\frac{A \vee B, \quad A \to B}{B}$	$\frac{A \vee B, \quad \neg A \vee B}{B}$

Правило резолюции логично и полно. Полнота и логичность резолюции могут быть легко доказаны. Здесь приводятся доказательства для случая исчисления высказываний.

Правило вывода *логично*, если заключение является логическим следствием посылок.

Теорема.
Правило резолюции логично.
Доказательство.

Введем обозначения. Пусть I – произвольная интерпретация, а C_1 и C_2 – это предложения, которые резольвируются по правилу резолюции, P – тот литерал, по которому ведется резольвирование. Пусть, для определенности, предикат P входит в предложение C_1 положительно, а в C_2 с отрицанием. Пусть C_1' и C_2' – это то, что получается после выбрасывания из предложений контрарных литералов.
По условию теоремы $I(C_1)$ & $I(C_2)$. Тогда если $I(P)$, то $C_2' \neq \varnothing$ & $I(C2')$, а значит $I(C_1' \vee C_2')$. Если же $I(\neg P)$, то $C_1' \neq \varnothing$ & $I(C_1')$, а значит $I(C_1' \vee C_2')$.
Ч.т.д.

Правило вывода *полно*, если любая общезначимая формула может быть выведена по этому правилу.

Теорема.
Правило резолюции полно.
Доказательство.
Как известно, правило modus ponens для исчисления высказываний полно и является частным случаем резолюции.
Ч.т.д.

4.2.2. Сведение к предложениям

Разберем, откуда берутся предложения. Существует несколько эквивалентных вариантов сведения формул исчисления предикатов к предложениям. Мы используем следующий. Берем любую формул исчисления предикатов и

1) элиминируем в ней импликацию, заменяя ее на дизъюнкцию с отрицанием первого операнда,
2) протаскиваем отрицания по правилам де Моргана к атомам,
3) разделяем связанные переменные так, чтобы они случайно не совпадали в разных кванторах,
4) элиминируем кванторы существования с помощью *сколемизации*, вводя новые функциональные символы и константы,
5) приводим к предваренной форме, когда у нас формула выглядит так: кванторная приставка – матрица, не содержащая кванторов,
6) элиминируем кванторы всеобщности, поскольку мы рассматриваем только замкнутые формулы,
7) приводим к конъюнктивной нормальной форме (то есть главной операцией является конъюнкция, и она соединяет группы дизъюнкций),
8) рассыпаем всю формулу на множество предложений, каждое предложение является бескванторной дизъюнкцией литералов.

Пример.
Сведем к предложениям известную формулу

$$\forall x\big(P(x) \to \exists y\, Q(x,y)\big)$$

1) $\forall x\big(\neg P(x) \vee \exists y\, Q(x,y)\big)$
2) формула без изменений
3) формула без изменений
4) $\forall x\big(\neg P(x) \vee Q(x, f(x))\big)$
5) формула без изменений

6) $\neg P(x) \vee Q(x, f(x))$
7) формула без изменений
8) формула без изменений

Пример.

Сведем к предложениям отрицание предыдущей формулы

$$\neg \forall x \left(P(x) \to \exists y \, Q(x, y) \right)$$

1) $\neg \forall x \left(\neg P(x) \vee \exists y \, Q(x, y) \right)$
2) $\exists x \left(P(x) \,\&\, \forall y \, \neg Q(x, y) \right)$
3) формула без изменений
4) $P(a) \,\&\, \forall y \, \neg Q(a, y)$
5) $\forall y \left(P(a) \,\&\, \neg Q(a, y) \right)$
6) $P(a) \,\&\, \neg Q(a, y)$
7) формула без изменений
8) $P(a), \quad \neg Q(a, y)$

4.2.3. Унификация

Для того чтобы применить метод резолюций, нам нужно найти контрарные литералы. В случае исчисления высказываний, такой проблемы нет. Любая переменная и та же переменная с отрицанием – это и есть пара контрарных литералов. В случае исчисления предикатов первого порядка это не так. Даже когда мы нашли два вхождения одного предиката, положительное и с отрицанием, это еще не значит, что найдена контрарная пара. После предикатной буквы могут стоять разные параметры, и они могут отличаться в том и в другом случае. В некоторых случаях путем подстановки можно литералы *унифицировать*, то есть сделать такие замены переменных, что литералы сделаются одинаковыми. В некоторых случаях этого сделать нельзя. Существует алгоритм, который проверяет, можно ли унифицировать две формулы (в частности, два атома) и находит набор унифицирующих подстановок, если унификация возможна.

```
Proc Unify (e1, e2)
     if atom(e1) ∨ atom(e2) then
          if e1=e2 then return nil
          elseif var(e1) then return{e2/e1}
          elseif var(e2) then return {e1/e2}
          else return(fail)
          end if
     end if
     f1:=car(e1); t1:=cdr(e1); f2:=car(e1); t2:=cdr(e2)
     z1:=Unify(f1,f2)
     if z1 = fail then return fail end if
     g1:=t1{z1}; g2:=t2{z1}; z2:=Unify(g1,g2)
     if z2 = fail then return fail end if
```

```
    return {z1, z2}
end proc
```

Данный алгоритм проверяет, существует ли для двух разных формул *общий частный случай*, то есть существует ли такой набор подстановок, который делает формулы одинаковыми.

Процедура вычисления общего частного случая для двух формул, если он существует, называется *унификацией*.

Набор подстановок каких-то термов вместо переменных в обе формулы, в результате чего они становятся одинаковыми, называется *унификатор*.

Кратчайший набор таких подстановок называется *наиболее общий унификатор* (*Least Common Unifier*).

Процедура, представленная выше, достаточно очевидна. Стандартные обозначения языка ЛИСП и списочные представления для формул являются общепризнанным программистским стандартом для области ИИ.

В частности здесь используются следующие стандартные функции – стандартная функция `atom`, которая проверяет, является ли переданный ей на вход список одноэлементным, функция `var`, которая проверяет, является ли переданное ей выражение именем переменной, стандартная функция `car`, которая возвращает первый элемент списка и стандартная функция `cdr`, которая возвращает список, содержащий все элементы, кроме первого.

4.2.4. Опровержение методом резолюций

Если мы возьмем некоторое множество посылок A, и возьмем отрицание формулы G, и все вместе это окажется несовместным, то есть если из этой совокупности формул выводимо тождественно ложное утверждение, то из этого следует, что формула G выводима из множества посылок A. Это типичная схема доказательства от противного.

Теорема.

Если $A,\neg G \vdash F$, *где* F *противоречие*, то $A \vdash G$.

Доказательство (для случая исчисления высказываний).

По теореме дедукции если $A,\neg G \vdash F$ то $\vdash A \,\&\neg G \rightarrow F$, и значит $A \,\&\neg G \rightarrow F$ – тавтология. Но $A \,\&\neg G \rightarrow F = \neg\,(\,A \,\&\neg G\,) \vee F = \neg A \vee G = A \rightarrow G$. Значит $A \rightarrow G$ – тавтология и $\vdash A \rightarrow G$, откуда, еще раз применяя теорему дедукции, имеем $A \vdash G$. **Ч.т.д.**

Отсюда вытекает следствие. Поскольку мы рассматриваем дизъюнкции, то пустые дизъюнкции следует рассматривать как тождественно ложные. А значит, в качестве невыполнимой формулы F, которая упомянута в предыдущей теореме, удобно использовать пустую дизъюнкцию, не содержащую никаких дизъюнктивных слагаемых. Именно так традиционно делается в методе резолюций, а пустая дизъюнкция традиционно обозначается $\square$.

Следствие.

Если $A,\neg G \vdash \square$, *где* $\square$ *– пустая формула*, то $A \vdash G$.

Окончательно метод резолюций формулируется следующим образом:

Метод резолюций – это метод автоматического доказательства теорем, основанный на опровержении множества посылок A и отрицания целевой теоремы G с использованием правила резолюции.

Метод резолюций применяется следующим образом. Берется совокупность формул A и каждая из них по отдельности сводится к предложениям. При этом обязательно производится разделение переменных, чтобы они случайно не совпадали. Потом берется целевая формула G, к ней применяется формальное отрицание, и после этого формула $\neg G$ сводится к предложениям. Образуется множество формул, каждая из которых является предложением. К ним систематическим образом применяется правило резолюции до тех пор, пока не получается пустая формула.

Возможны три исхода.

1. В результате применения метода резолюций получается пустая формула. Это означает, что теорема доказана. Мы построили вывод формулы G из формул A.
2. В результате применения метода резолюций не получается пустая формула, и больше применить правило не удается, потому что нет резольвируемых предложений, которые давали бы новые предложения. Это означает, что мы опровергли теорему. Мы показали, что формула G невыводима из множества формул A.
3. Правило резолюции применяется, множество предложений пополняется, среди них нет пустых, и есть резольвируемые. То есть процесс идет и конца ему не видно. Это ничего не означает.

Другими словами, метод резолюций является частично корректной процедурой. Третий исход нельзя назвать зацикливанием, поскольку нет никакого способа определить, почему метод резолюции продуцирует новые резольвенты, но при этом ничего не происходит. Означает ли это, что теорема верна, но мы просто не дождались завершения? Не известно. Означает ли это, что в результате так и не получится пустой формулы? И этого мы тоже не знаем. Это логически неразрешимая задача. В принципе нельзя узнать, как долго нужно ждать, для того, чтобы получить ответ на этот вопрос.

4.2.5. Программная реализация метода резолюций

Все сказанное выше можно записать в виде программы на псевдокоде.

```
Proc R(S)
     C := S
M:   if □ ∈ C then return OK end if
     Select(C, p1, p2, l1, l2, s)
     if p1,p2 =nil then return fail end if
     p := Res(p1, p2, l1, l2, s)
     C := C + p;
     go to M
end proc

Proc Res (p1, p2, l1, l2, s)
```

```
    p1 := p1{s}; p2 := p2{s}
    p1 := p1 – l1; p2 := p2 – l2
    return p1 ∨ p2
end proc

Proc Select(C, p1, p2, l1, l2, s)
Выбрать в множестве предложений C два предложения p1, p2,
содержащие контрарные литералы l1, l2, унифицируемые
наиболее общим унификатором s
end proc
```

При записи этих процедур используется стандартная практика при составлении алгоритмов – то, что можно фиксировать, фиксировано. В частности, процедура `Res` фиксирована. Нефиксированной процедурой является процедура `Select`. Стратегия управления перемещена в эту функцию, которая выбирает два предложения для резольвирования.

Пример.

Всякий программист знает метод резолюций.

Студенты не знают метода резолюций.

Некоторые студенты умные люди.

Следовательно, некоторые умные люди не являются программистами.

Логически последнее утверждение верно. Механическим путем с помощью метода резолюций это утверждение можно проверить следующим образом.

1. $\neg$ П(x) $\vee$ Р(x)
2. $\neg$ С(x) $\vee \neg$ Р(x)
3. С(a)
4. У(a)
5. $\neg$ У(x) $\vee$ П(x)
6. П(a) (4,5)
7. Р(a) (1,6)
8. $\neg$ С(a) (7,2)
9. $\square$ (8,3)

Поясним, как получаются формулы в этом формальном выводе.

Первое предложение получено трансляцией первого исходного утверждения на естественном языке в форму предложений. При этом П означает «быть программистом», Р означает «знать метод резолюций».

Второе предложение аналогично получается из второго утверждения.

Элиминация существования в третьем исходном утверждении, то есть сколемизация, привела к появлению неопределенной константы a, а конъюнкция, которая там была, рассыпалась на шаге 8 сведения к предложениям. Поэтому из утверждения «Некоторые студенты умные люди» получается два исходных предложения: С(a) и У(a). 5 пункт получился путем отрицания цели: «некоторые умные люди не являются программистами». Формально это записывается так:

$$\exists x \; У(x) \; \& \; \neg П(x).$$

К этому прибавляется спереди отрицание. Существование превращается во всеобщность и И превращается в ИЛИ. Дальше 4 и 5 резольвируются по У, с подстановкой *a* вместо *x*, берется 1 и 6 и резольвируются по П с подстановкой *a* вместо *x*. Дальше берутся предложения 7 и 2 и резольвируются по Р. Получается ¬C(*a*), затем 8 и 3 предложения резольвируются по С. В результате получаем пустую дизъюнкцию. Утверждение доказано.

4.3. Стратегии поиска опровержения методом резолюций

Один из самых важных вопросов, касающихся МР – откуда взять функцию `Select`? Есть несколько примеров стратегии – способов выбора пары резольвирумых предложений.

Способ выбора предложений для резольвирования называется *стратегий метода резолюций.*

4.3.1. Полные и неполные стратегии

Самое важное понятие, относящееся к стратегиям – это идея полноты.

Если существует какое-либо опровержение по МР, то существует опровержение по данной стратегии. Такая стратегия называется *полной.* И стратегия называется *неполной*в противном случае.

Неполнота стратегии означает, вообще говоря, что утверждение доказать можно, но с помощью выбранной стратегии не удается. Тем не менее, такие стратегии достаточно часто применяется, потому что неполные стратегии зачастую являются более эффективными, чем полные стратегии. С помощью неполной стратегии в некоторых случаях быстрее получают вывод. Правда, иногда неполные стратегии его вообще не получают, но зато, когда они срабатывают, они срабатывают быстрее. Кроме того, неполная в общем случае стратегия может оказаться полной, если мы ограничим исходный класс предложений.

4.3.2. Примеры стратегий МР

Сначала рассмотрим две полные стратегии.

1. Первая стратегия – хорошо известная стратегия *поиска в ширину* или *полный перебор*. В этом случае на каждом шаге строятся все возможные резольвенты.

2. *Линейная по входу стратегия*: в этом случае одно из родительских предложений берется из основного (исходного) множества предложений, а второе – последнее полученное предложение. Т.е всегда одно предложение берется из исходного множества, а второе на первом шаге из того же исходного множества, а потом может быть из последнего построенного.

Примерами стратегий, неполных в общем случае, которые, тем не менее, заслуживают особого внимания, поскольку применяются чаще всего, являются следующие:

1. *Единичная положительная* резолюция (*поиск от данных*). В этом случае одно из родительских предложений – положительный литерал (факт).

2. *Входная отрицательная* резолюция (*поиск от цели*). При этом одно из родительских предложений – отрицательный литерал (цель).

Примеры применения полных стратегий.

Рассмотрим полный перебор на примере задачи о программистах, которые знают или не знают МР (см. параграф 4.2.5).

1. Полный перебор (поиск в ширину).

В результате полного перебора получается довольно много предложений:

$\neg П(x) \vee Р(x)$	$\neg С(x) \neg \vee Р(x)$	$С(a)$	$У(a)$	$\neg У(x) \vee П(x)$
$\neg П(x) \neg \vee С(x)$	$\neg Р(a)$	$П(a)$	$Р(x) \neg \vee У(x)$	
$\neg С(x) \neg \vee У(x)$	$\neg С(a) \neg П(a)$	$\neg У(a)$	$Р(a)$	
… … …	□			

Первая строчка – это исходные пять предложений. Во второй строчке написан полный перебор всех предложений, которые получаются путем резольвирования первой строчки (исходных предложений). В третьей строчке написаны все новые предложения, которые получаются путем резольвирования тех предложений, которые написаны в первых двух строчках. В четвертой строчке появится пустой квадратик – пустое предложение. Но на самом деле там довольно много предложений, они не выписаны, потому что не помещаются в одну строчку.

В результате после третьего пополнения исходного множества предложений будет построено пустое предложение и закончен вывод, но перед этим будет построено довольно много предложений, которые в выводе не участвуют, они, безусловно, лишние. Это и есть полная стратегия, полный перебор. Она не эффективна, как и большинство полных стратегий, однако всегда заканчивается положительным для нас результатом – мы получаем ответ, если он существует.

2. Линейная по входу стратегия (поиск в ширину).

Линейная по входу резолюция выглядит для той же задачи более эффективной, чем полный перебор.

$\neg П(x) \vee Р(x)$	$С(x) \neg \vee Р(x)$	$С(a)$	$У(a)$	$\neg У(x) \vee П(x)$
$\neg П(x) \neg \vee С(x)$	$\neg Р(x)$	$П(a)$	$Р(x) \neg \vee У(x)$	
$\neg П(a)$	$\neg С(x) \neg \vee У(x)$	$\neg С(a)$	$Р(a)$	$\neg У(a)$ $\neg С(a)$

□

В этом случае мы все время резольвируем один из фактов, который был вначале. Здесь выписаны все предложения, которые получаются по ходу решения. Их несколько меньше, чем в случае полного перебора. Но в этом случае пустое предложение появится только на четвертом уровне пополнения. Однако это ничего не значит. В другом случае полный перебор может оказаться эффективнее.

Естественно, есть и другие примеры полных стратегий, которые широко представлены в литературе.

4.4. Извлечение результата

Метод резолюций строит доказательство. С помощью специальных приемов из доказательства можно извлекать ответы (знания) любого типа.

4.4.1. Извлечение результата (да/нет)

Стратегии резолюции, которые мы рассмотрели, пока давали ответ на вопрос – выводимо или не выводимо решение. То есть, получался бинарный ответ на вопрос – да или нет.

Пример. Рассмотрим следующие утверждения.

Все студенты группы 5057 знают метод резолюций.
Власенко студент группы 5057
Знает ли Власенко метод резолюций?

При переводе в форму предложений и резольвировании получаем следующий вывод.

1. ¬ 5057(x) ∨ P(x)
2. 5057(Власенко)
3. ¬ P(Власенко)
4. P(Власенко) (1,2)
5. □ (3,4)

Ответ – да.

Все студенты группы 5057 знают метод резолюций.
Халепский не знает метод резолюций.
Халепский является студентом группы 5057?

1. ¬ 5057(x) ∨ P(x)
2. ¬ P(Халепский)
3. ¬ 5057(Халепский)
4. ¬ 5057(Халепский) (1,2)

Ответ– нет.

Это и есть извлечение результата. С помощью выбранной стратегии мы смогли получить ответ на поставленный вопрос.

4.4.2. Извлечение результатов (факты)

А теперь попробуем поставить вопрос так, чтобы получить ответ не форме да-нет, а в форме факта. Для этого нужно модифицировать метод резолюций введением специального предиката – традиционно называемого предикатом ANS. Предикат ANS – это предикат, в котором накапливается значение ответа. Считается, что этот предикат эквивалентен пустому предложению. Если есть предложение, содержащее только предикаты ANS, то это пустое предложение. Так модифицируется метод резолюций.

Пример.

Все студенты группы 5057 знают метод резолюций.
Власенко студент группы 5057.
Королев студент группы 5057.
Кто знает метод резолюций?

Теперь целевое предложение P(x) выглядит так: если P(x), то ANS(x), то есть если x знает метод резолюций, то x является ответом. В формальной записи в форме предложений вопрос имеет вид ¬ P(x) ∨ ANS(x). Напоминаем, что с точки зрения управления выводом ANS = □.

Мы проводим резолютивный вывод и получаем в 6 предложений, последнее из которых ANS(Власенко), то есть получаем ответ:

1. ¬ 5057(x) ∨ P(x)
2. 5057(Власенко)
3. 5057(Королев)
4. ¬ P(x) ∨ ANS(x)

5. ¬ 5057(x) ∨ ANS(x) (1,4)
6. ANS(Власенко) (5,2)

Можно остановиться, а можно продолжать построение новых резолютивных выводов, как и делают системы логического программирования, основанные на языке Пролог. Тогда получатся все ответы, которые можно получить из условий задачи.

7. ANS(Королев) (5,3)

4.4.3. Извлечение результатов (термы)

Используя технику предикатов ANS, можно получать не только факты, но и результаты процедурного уровня. Основываясь на этом, можно синтезировать выражение (терм), вычисляющее некоторую функцию.

Пример.

Имеется отношение отец и дед. Аксиома, описывающая эту ситуацию, заключается в следующем: у любого человека есть отец, и отец отца – это дед.

$O(x, y) = x$ отец y
$D(x, y) = x$ дед y
$\forall x,y,z\ O(x, y)\ \&\ O(y, z) \rightarrow D(x, z)$
$\forall y \exists x\ O(x,y)$.

Дальше строим вопрос – кто является отцом x?

$? D(y, x) \rightarrow ANS(y)$

В результате метода резолюций, получается ответ:

1. $\neg O(x, y) \vee \neg O(y, z) \vee D(x, z)$
2. $O(f(y), y)$
3. $\neg D(y, x) \vee ANS(y)$
4. $\neg O(y, z) \vee D(f(y), z)$ (1,2) $f(y)/x$
5. $D(f(f(y)), y)$ (4,2) $f(y)/y$, y/z
6. $ANS(f(f(x)))$ (3,5)

где f – это функция вычисления отца, которая была получена при сколемизации второй аксиомы этого примера.

Таким образом, метод резолюций с точки зрения представления знаний является вполне адекватным средством. Он позволяет получать содержательные ответы на содержательные вопросы.

4.5. Хорновский случай

Существует подкласс предложений, который очень важен с точки зрения практического применения метода резолюций.

4.5.1. Хорновские предложения

Давайте попробуем синтаксически описать общие свойства следующих предложений:

A	Факты
A1&…&AN→B	Правила
B	Цели

После перевода в форму предложений эти формулы имеет такой вид:

A

$\neg A1 \vee ... \neg AN \vee B$

$\neg B$

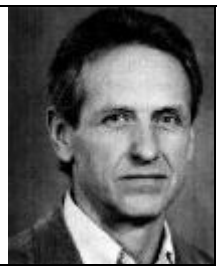
Хорн в свое время заметил, что три эти формулы в форме предложений обладают следующей синтаксической характеристикой – в них содержится не более одного положительного литерала. Выше приведены все три возможных случая: один положительный литерал, один положительный литерал и несколько отрицательных, нет положительных литералов (одни отрицательные).

Предложения, содержащие не более одного положительного литерала, называются *хорновскими*.

Кроме этого, множество таких предложений (хорновских формул) замкнуто относительно правила резолюции. Если мы резольвируем хорновские предложения, мы снова получаем хорновское предложение, потому что мы вычеркиваем один положительный и один отрицательный литерал, в сумме оказывается не более одного положительного литерала.

Подход для применения метода резолюций в программировании был разработан не Хорном, это было уже сделано позднее Р. Ковальски. Оказалось, что метод поиска от целей и метод поиска от данных, которые не полны в общем случае, полны в хорновском случае. Если существует какой-либо вывод, то существует вывод от целей для хорновского случая. Однако в общем случае это не так.

Ковальский Роберт *(Kowalski Robert)*.
Совместно с Аланом Колмероером разработал PROLOG (programmation en logique) – логический язык программирования.



Основываясь на хорновских случаях, был разработан язык Пролог, который иногда называют *логическим программированием*. Однако понятия Пролога и логического программирования не следует отождествлять. Язык Пролог – это маленький частный случай, а именно это реализация метода резолюций для хорновского случая со стратегией «входная отрицательная». Можно использовать другую стратегию, можно взять другой подкласс, можно взять вообще исчисление не первого порядка, и все это будет логическое программирование, но не будет являться Прологом.

4.5.2. Замечания по реализации

Разработать прямолинейную систему логического программирования достаточно легко. Однако все такие наивные реализации оказываются практически бесполезными. Это происходит из-за того, что не учитывается несколько важных факторов.

1. Важен не выбор стратегии резольвирования, важно качество процедуры унификации. Приведенная выше процедура унификации совсем не эффективна. Дело в том, что процедура унификации проверяет, унифицируемы ли два литерала. И если они унифицируемы, выдает подстановку, которая их унифицирет. Более

90% машинных ресурсов системы логического программирования при этом уходит на унификацию.

2. Большинство программистов пытается написать программу унификации, которая как можно скорее выдает ответ – набор подстановок. А в данном случае (для унификации) требуется создать программу, которая как можно скорее *не* получала бы ответ, потому что большинство литералов *не* унифицируемы. Нужно как можно скорее выдать fail, оборвать выполнение процедуры, не нужно вычислять ответ. Его, скорее всего, вычислять не придется. Это требует тонких программистах хитростей.

3. Большинство операций, которые встречаются в реальных задачах, обладают специальными свойствами – коммутативностью, транзитивностью и т.д. Если их не использовать, то окажется, что не унифицируемо то, что на самом деле, по существу, унифицируемо. А если их использовать на логическом уровне, то количество аксиом разрастется с коэффициентом, который равен величине аксиоматики таких теорий, как теория транзитивности или коммутативности. То есть все увеличивается в несколько раз. Конечно, учет таких вещей, как транзитивность и коммутативность, нужно делать на уровне унификации термов, а не на уровне логики.

Тема 5. Системы дедукции на основе правил

В данной теме мы попробуем на конкретном примере рассмотреть совместно различные способы представления знаний, которые мы разобрали, то есть представления знаний на основе системы продукции и представления знаний на основе исчисления предикатов первого порядка. На самом деле между ними нет никакой непреодолимой границы. Это все сводимые друг к другу вариации на одну и ту же тему, что дает повод считать, что действительно есть предмет «Системы представления знаний». Если бы это были абсолютно разрозненные вещи, то было бы совершенно непонятно, почему они объединяются в рамках одного курса. Мы сейчас легко убедимся, что они совсем не разрозненные.

При этом за рамками данной книги остается аналогичная ситуация, которая имеет место и в других случаях – например, если мы используем семантические сети или фреймы. Здесь тоже есть некое общее ядро, есть некая содержательная часть, которую можно выявить, как общее содержание теории представления знаний.

Сначала мы разберем, что же является мотивацией в попытках объединить разные системы представления знаний. Затем еще раз с других позиций рассмотрим форму И/ИЛИ, которая уже дважды встречалась в данном учебнике под разными названиями: первое название было «разложимые системы продукции», а второе – «поиск на игровых деревьях». Мы еще раз рассмотрим этот аспект системы представления знаний, и на трех примерах попытаемся осознать, что во всем этом есть общее содержание, есть глубокие внутренние взаимосвязи между приемами и техниками, которые здесь используются. В конце этой главы будет продемонстрирована одна конкретная техника, которая называется система дедукции на основе правил.

5.1. Недостатки метода резолюций

Прежде всего, разберем недостатки, присущие методу резолюций. У этого метода есть две существенные имманентно присущие особенности, которые могут рассматриваться в некоторых случаях как недостатки.

5.1.1. Потеря импликативности

Первая особенность состоит в том, что метод резолюции работает с приложениями – с бесквантворными дизъюнкциями литералов. В этих дизъюнкциях нет направления. На самом деле то, в каком порядке буквы находятся в предложении очень важно, потому что все алгоритмы линейно просматривают информацию и анализируют ее в том порядке, в котором буквы попадаются в формулах. Этот порядок может быть важен.

Если сравнить метод резолюций с привычным методом доказательства на основе правила Modus ponens, то в самом это правиле заложено явное направление рассуждений. Правило Modus ponens выглядит следующим обоазом.

$$\frac{A,\quad A \to B}{B}$$

Понятно, что в этом случае нужно начинать с A, и тогда мы придем к B. Есть явное направление рассуждений, заданное самой стрелкой. Но когда мы элиминируем

стрелки импликаций при сведении формул к предложениям, это направление может быть потеряно.

Пример.

$(\neg A \wedge \neg B) \rightarrow C$ $\qquad$ $A \vee B \vee C$

$(\neg B \wedge \neg C) \rightarrow A$ $\qquad$ $\text{A} \vee \text{B} \vee \text{C}$

Выше приведены два совершенно разных импликативных правила, которые отражали разную семантику, а после преобразования к предложениям эта разница исчезла. Метод резолюции не знает, с чего нужно начинать и куда нужно двигаться. У него все буквы одинаковы, он их просто механически теребит, пытаясь вывести пустое предложение, но в какую сторону надо двигаться, чтобы вывести пустое предложение – этой информации нет, она утрачена, хотя, возможно, присутствовала в исходной постановке задачи.

Постановки задачи для системы автоматического доказательства – это дело эксперта, который наполняет базу знаний. Он-то знает, в какую сторону нужно использовать импликацию, и мог бы системе подсказать. Например, эксперт может знать, что вместо прямого правила $\neg A \rightarrow B$ следует использовать контрапозитивную форму $\neg B \rightarrow A$. Но при сведении к предложениям подсказки забываются. Метод резолюций видит только формулу $\text{A} \vee \text{B}$. Это первый недостаток метода резолюций. Называется это свойство – *потеря импликативности*.

5.1.2. Размножение литералов

Второй недостаток данного метода заключается в следующем. По дороге преобразования в предложения мы вынуждены использовать конъюнктивную форму. При этом формула может расширяться. В методе резолюций может оказаться (и часто в худшем случае оказывается), что коэффициент расширения, который здесь наблюдается, носит экспоненциальный характер. Другими словами может произойти так называемое *размножение литералов*.

Пример.

> Власенко отлично учится в группе 5057. Все, кто учится в данной группе, являются студентами, все студенты знают метод резолюций. Отличники умные. Попробуем показать следующий факт: существует студент, который знает метод резолюций и является либо программистом, либо умником.

Введем соответствующие предикаты и запишем формулы.

0. $\exists x\ \text{Р}(x) \wedge (\text{П}(x) \vee \text{У}(x))$ цель
1. 5057(Власенко) факт
2. О(Власенко) факт
3. $\forall x\ 5057(x) \rightarrow \text{С}(x)$ правило
4. $\forall x\ \text{С}(x) \rightarrow \text{Р}(x)$ правило
5. $\forall x\ \text{О}(x) \rightarrow \text{У}(x)$ правило

Начинаем преобразовывать. По правилу метода резолюций к цели спереди надо приписать отрицание. При этом приписывании первое утверждение разваливается на два. И буква Р, которая в исходной постановке присутствовала один раз, после преобразования появляется два раза. А теперь начинаем резольвирование.

1. $\neg\text{Р}(x) \vee \neg\text{П}(x)$
2. $\neg\text{Р}(x) \vee \neg\text{У}(x)$

3. $\neg C(x) \vee P(x)$
4. $\neg 5057(x) \vee C(x)$
5. 5057(Власенко)
6. $\neg(x)O \vee У(x)$
7. О(Власенко)
8. $\neg П(x) \vee \neg C(x)$ 1, 3
9. $\neg П(x) \vee 5057(x)$ 8, 4
10. $\neg П(x)$ 9, 5
11. $\neg У(x) \vee \neg C(x)$ 2, 3
12. $\neg У(x) \vee \neg 5057(x)$ 11, 4
13. $\neg$У(Власенко) 12, 5
14. $\neg$О(Власенко) 13, 6
15. □ 14, 7

Пункты 8, 9 и 10 – это результативные выводы, резольвенты, полученные самым простым образом. Берем два первых подходящих предложения и начинаем резольвировать. 8, 9 и 10 пункты в этом примере – это первые подходящие, и легко сообразить, что это полученное $\neg П(x)$ нам совсем не нужно. Его потом некуда будет девать. Программисты вообще в исходных данных не упоминались. Это бессмысленно потраченная машинное время и память на кусок вывода. Начинать надо было сразу с пункта 11 (резольвируя предложения 2 и 3), первое предложение не надо было рассматривать. Вот эта формула $\neg P(x)$, которая появилась в предложении 1, произошла в результате размножения литералов и, безусловно, вредна для решения задачи.

Вывод построен, но по дороге был ложный ход, который вызван тем, что произошло размножение литералов, вытекающее из преобразования в конъюнктивную нормальную форму. В результате количество вхождений литералов в предложения увеличилось. В худшем случае, такое увеличение может носить экспоненциальный характер.

5.1.3. Прямая систем дедукции

Теперь на ту же картинку посмотрим с другой стороны. Не будем сводить все к предложениям, применим обычное правило Modus ponens. Не будем использовать доказательство от противного, применим прямую систему дедукции. В результате все получается гораздо лучше, проще и быстрее.

0. $\exists x\, P(x) \wedge (П(x) \vee У(x))$ цель
1. 5057(Власенко) факт
2. О(Власенко) факт
3. $\forall x\, 5057(x) \rightarrow C(x)$ правило
4. $\forall x\, C(x) \rightarrow P(x)$ правило
5. $\forall x\, O(x) \rightarrow У(x)$ правило
6. С(Власенко) Modus ponens 1, 3
7. Р(Власенко) Modus ponens 6, 4
8. У(Власенко) Modus ponens 2, 5
9. П(Власенко) $\vee$ У(Власенко) Введение дизъюнкции $\vee$+, 8
10. Р(Власенко) $\wedge$ (П(Власенко) $\vee$ У(Власенко))

Введение коньюнкции &+, 7, 9

11. $\exists x$ Р(x) $\wedge$ (П(x) $\vee$ У(x)) Введение существования $\exists$+, 10

Предложения 1 и 3 по Modus ponens дают С(Власенко), 6 и 4 дают Р(Власенко), 2 и 5 дают У(Власенко), и, применяя правила, аналога которым в методе резолюций нет, но которые, очевидно, логичны, получаем цель. Прямая система (обычное рассуждение, а не метод доказательства от противного) – простое рассуждение по правилу Modus ponens и введение дополнительных связок, гораздо быстрее, чем в первом случае позволило получить тот же самый ответ.

Вот в этом и состоит основная идея. Мы хотим сохранить преимущества, которые нам дает представление знаний с помощью предложений и избавиться от недостатков, которое дает это представление. Избавиться от разложения литералов и сохранить направление импликации, чтобы иметь возможность проводить такие эффективные и короткие выводы, как во втором случае по сравнению с неэффективными и длинными выводами в первом случае.

5.2. Форма И/ИЛИ

Попробуем преобразовать исходные формулы, с помощью которых записано наше знание о предметной области (формулы исчисления первого порядка) не в форму предложений, а в так называемую *форму И/ИЛИ*, просто проводя меньшее количество шагов преобразования. Так же, как при сведении к предложениям в методе резолюций элиминируем импликацию, протаскиваем отрицание, элиминируем квантор существования, строим предваренную форму, убираем квантор всеобщности и разделяем переменные в главных конъюнктах.

Если вы обратили внимание, пропущен этап построения конъюнктивной формы. Какая форма была, такая форма и осталась, мы только избавились от кванторов и разделили переменные. Легко видеть, что в результате элиминации импликации, протаскивания отрицания, элиминации квантора существования, построения предваренной формы, элиминации всеобщности и переименования переменных, количество литералов сохраняется.

Пример. Дана формула (рис. 32).

$$\exists u \, \forall v \, (\mathrm{Q}(v, u) \wedge \neg \, ((R(v) \vee P(v)) \wedge S(u, v)).$$

Для использования формы И/ИЛИ предлагается представить формулу в виде гиперграфа следующим образом. Там, где главная операция $\wedge$, ставится две раздельных 1-дуги, там, где главная операция $\vee$, ставится одна 2-дуга. Ход рассуждения достаточно прост. Когда известно, что $A \wedge B$, то можно повести рассуждение в одну сторону, используя то, что дано A, а можно в другую, используя то, что дано B. Когда дано $A \vee B$ мы не знаем, в какую сторону конкретно двигаться, нужно двигаться сразу в обе стороны.

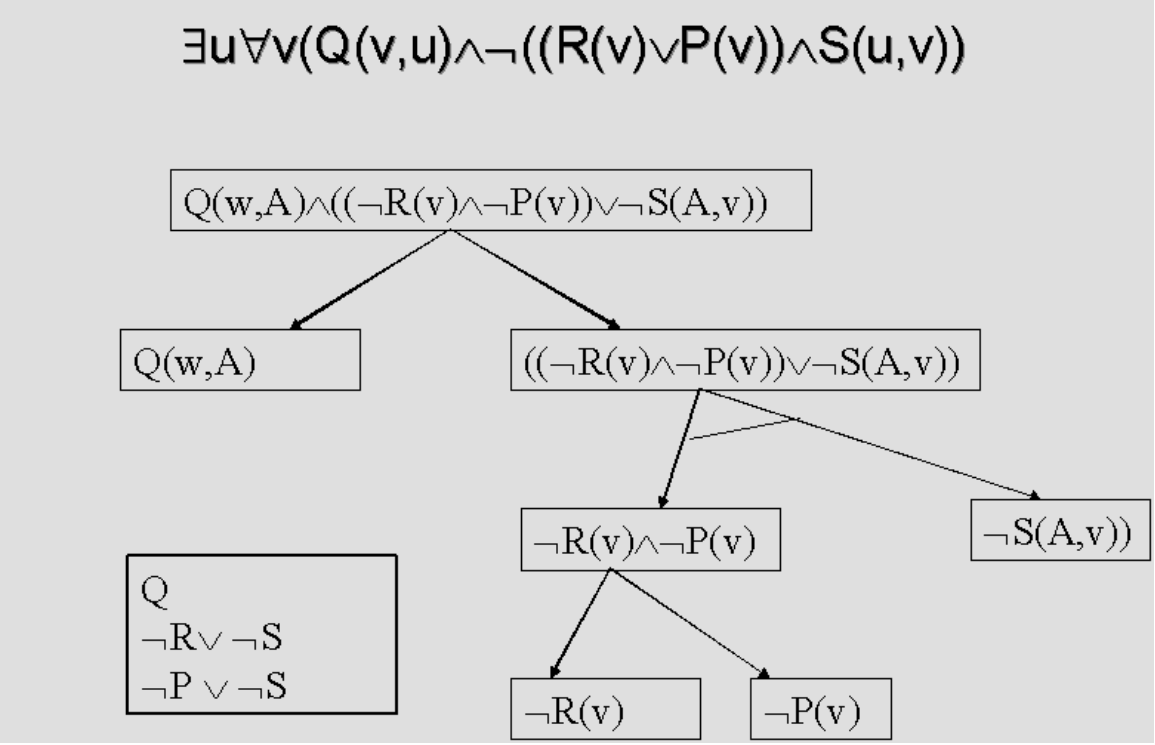


Рис. 32. Представления формы И/ИЛИ в виде гиперграфа

Очевидно, что множество предложений, в которые преобразуется корневая формула, может быть прочитано, как множество концевых вершин всех графов решения, которые есть на этом гиперграфе (рис. 32). На этом гиперграфе есть три графа решения. Один идет из корня в *Q*, второй идет из корня на следующий уровень и дальше обязательно завязывается на *S*, потому что там 2-дуга. Нас интересуют концевые вершины. Если исходную форму свести к предложениям, то их получится три: Q, $\neg R \vee \neg S$, $\neg P \vee \neg S$.

Заметим это обстоятельство – форма И/ИЛИ хранит то же информацию, которой мы располагали в методе резолюций, а также еще и другую информацию.

5.3. Прямая система дедукции на основе правил

Рассмотрим следующий способ построения машины доказательства, ядра интеллектуальной системы логического вывода в форме системы дедукции на основе правил.

5.3.1. Факты, правила и цели

Так же, как это принято в логическом программировании, мы должны разделить все множество формул, всю информацию, которая у нас есть в данной предметной области на три класса – факты, правила и цели.

При этом накладываются следующие синтаксические ограничения: То, что мы будем считать фактами, на самом деле ничем не ограничено, и может быть представлены в любой форме И/ИЛИ. Форма И/ИЛИ может быть получена по любой формуле исчисления предикатов. То, что мы будем считать правилами, обязательно должно иметь вид: $L \rightarrow W$, где L – литерал, а W – любая формула, которая будет представлена в форме И/ИЛИ.

Третье ограничение состоит в том, что цель должна быть дизъюнкцией литералов, то есть должна быть предложением. Другими словами, можно сформулировать условие остановки: целевая формула является предложением, граф решения заканчивается в целевых литералах.

5.3.2. Наращивание графа И/ИЛИ с помощью правил

Смысл этой процедуры прост. Возьмем факт или множество фактов и представим их в форме И/ИЛИ. Получиться гиперграф. Начнем к листовым узлам этого графа прицеплять правила. Каждый листовой узел построенного графа унифицируется с левой частью правила. Если унификация успешна, мы наращиваем граф (или дерево) И/ИЛИ, пока в этом графе не появится граф решения, листья которого, попадают в целевые узлы (то есть унифицируются с целевыми литералами). Когда происходит отождествление левой части правила с листовым узлом в графе, возможно, потребуется выполнение некоторых подстановок для унификации. Понятно, что если в разных графах решения применяются разные правила, то они применяются независимо. Но в одном графе решения невозможно одновременно подставить вместо одной и той же переменной разные термы. Значит, такие случаи должны быть исключены. Фактически механизм, который сейчас описывается, является частным случаем метода резолюций, который проводится с помощью перестроения графа.

Рассмотрим это на примере.

Пример.

Факт: Власенко умный и знает метод резолюций или он не студент

$$\neg C(B) \vee (P(B) \wedge У(B))$$

Правило 1: Все учащиеся группы 5057 являются студентами

$$\neg C(x) \rightarrow \neg 5057(x)$$

Правило 2: Тот, кто знает метод резолюций, является отличником

$$P(y) \rightarrow O(y)$$

Цель: Существует некто, кто или не учится в группе 5057 или является отличником

$$\exists z \neg 5057(z) \vee O(z)$$

На примере правила 1 видно преимущество прямой системы на основе правил. Вот почему сохранение импликативности так важно и полезно. Конструктор системы представления знаний для конкретной предметной области правило «все учащиеся группы 5057 являются студентами» может записать сразу в контрапозитивном виде: «если он не студент, то он не учится в группе 5057». Тем самым конструктор системы представления знаний (человек) дает подсказку программе логического вывода: в дедуктивных системах рассуждать выгоднее от общего к частному, а не от частного к общему. Именно это и сделано в правиле 1.

Мы строим прямое дедуктивное рассуждение, а не используем метод доказательства от противного. В отличие от метода резолюций, отрицание к цели применять не следует. Цель должна использоваться в том виде, в котором находится в формулировке задачи. В данном случае видно, что после сколемизации цель будет бесквантворной дизъюнкцией литералов, что и требуется в синтаксическом условии прямой системы дедукции.

Ход рассуждения таков (рис. 29).

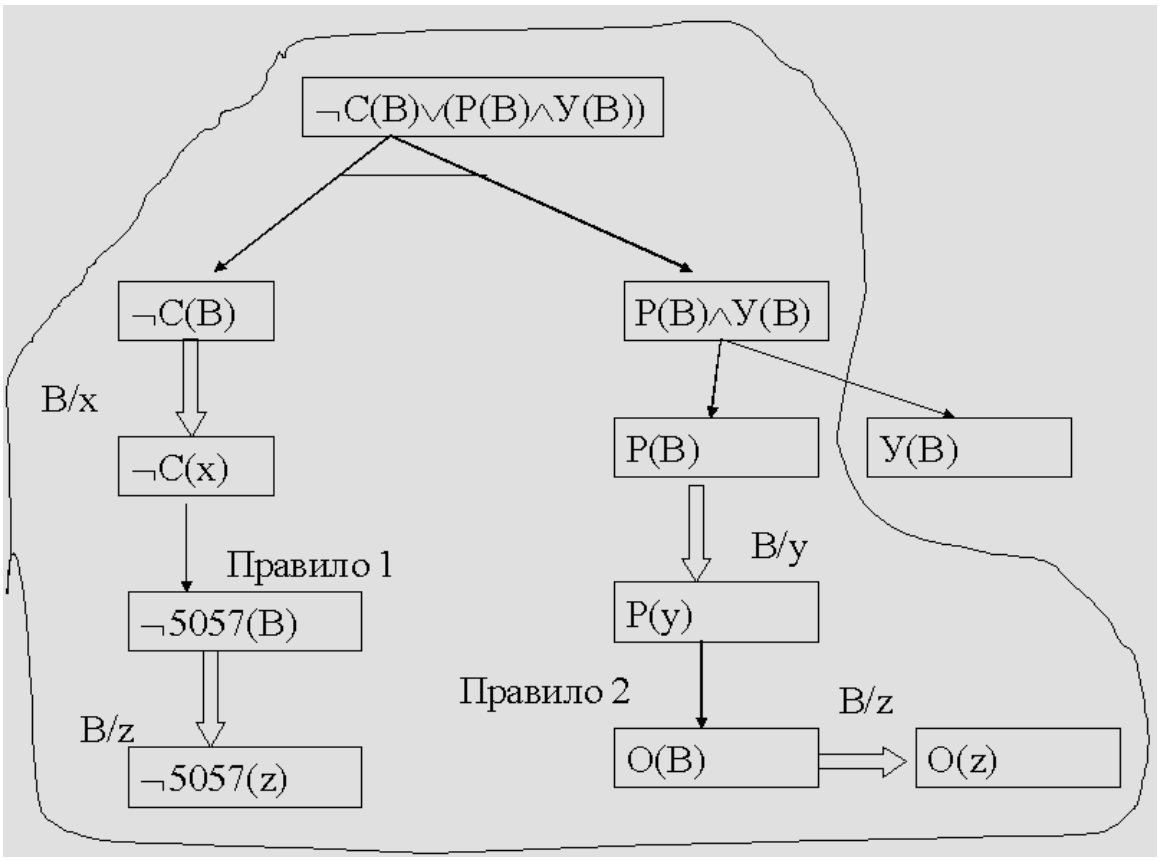


Рис.29. Расширение формы И/ИЛИ применением правил.

В начальный момент граф фактов имеет следующий список листовых узлов: ¬C(B), P(B) и У(B). Используем наше правило 1 для наращивания дерева. Левая часть правила унифицируется с подстановкой «B» вместо *x*. Это еще не решение, поскольку мы не построили граф решения, который заканчивается целевыми литералами. Поэтому продолжаем применять правила. Срабатывает правило 2, и результат отождествляется со вторым целевым литералом.

Построен граф решения, который заканчивается целевыми литералами (на рисунке от обведен).

Еще один **пример**.

Дан факт: P(x)∨Q(x)

Правило 1: P(A)→R(A)

Правило 2: Q(B)→R(B), где A и B – конкретные константы.

Цель: R(A)∨R(B), причем цель не является логическим следствием фактов.

Этот пример объясняет, в чем значения ограничения, говорящего о том, что подстановки на дугах соответствия должны быть согласованы (рис. 33).

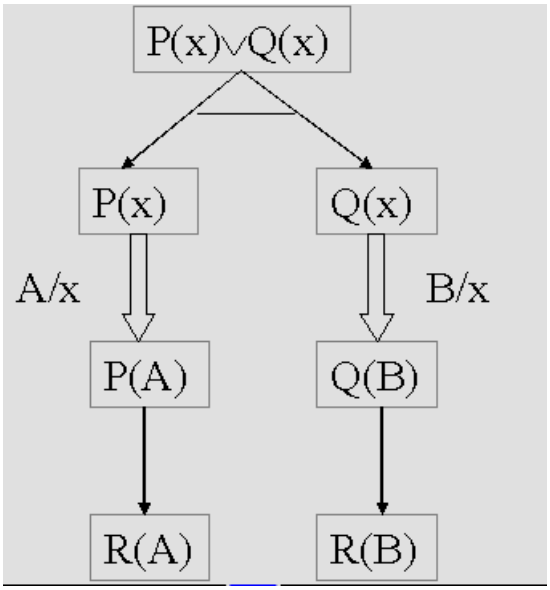


Рис. 33. Несогласованные подстановки на дугах соответствия.

Видно, что подстановки не совместимы, и нужно одновременно подставить A вместо *x* и B вместо *x*. В результате подстановки не согласованы, граф решения не построен, как и должно быть, потому что цель не является логическим следствием фактов и правил..

5.4. Обратная система дедукции на основе правил

Разобранный пример был достаточно занимательный. Мы провели в технике графов резолютивный вывод, но применимость этого приема ограничена тем, что у нас есть синтаксические ограничения на вид фактов, синтаксические ограничения на вид правил и синтаксические ограничения на вид целей.

5.4.1. Двойственный метод резолюций

Напомним известные определения и факты.

Если $f(x_1, \ldots, x_n)$ – булева функция, то булева функция

$$f^*(x_1, \ldots, x_n) = \neg f(\neg x_1, \ldots, \neg x_n)$$

называется *двойственной булевой функцией*.

Пользуясь тем фактом, что дизъюнкция и конъюнкция двойственны, а отрицание самодвойственно, наряду с методом резолюции можно построить *двойственный метод резолюции*, где элементами, подобными предложениям, будут не дизъюнкции литералов, а конъюнкции литералов. В двойственном методе резолюций сколемизации будет подвергаться не существование, а всеобщность, и будет использоваться не конъюнктивная нормальная форма, и дизъюнктивная нормальная форма (*k*-связка для конъюнкции, 1-связка для дизъюнкции). Главной операцией будет дизъюнкция. В двойственном методе резолюции мы будем отрицание целевой формулы вместе с исходными приводить к тождественно истиной формуле, которая тоже будет иметь пустую форму, потому что пустая конъюнкция является тождественно истинной.

Очевидные технические детали этих формальных логических построений мы опускаем, потому что нашей целью в этой главе является не работа с логическими формулами, а работа с графами И/ИЛИ.

5.4.2. Обратные правила

Теперь мы можем построить тоже синтаксически ограниченную, но с другой стороны, систему дедукции на основе правил. Можно идти от фактов к цели, а можно, и иногда это удобнее, идти от целей обратно к фактам по *обратным правилам*. В результате мы видим, что теперь у нас не ограничена форма целей – это любая форма И/ИЛИ. У нас ограничена форма правил, но совершенно другим образом. В правилах теперь правая часть должна быть литеральной, а левая может быть сколь угодно сложной. Кроме этого, ограничена форма фактов. Это должна быть, конечно, конъюнкция литералов, а не дизъюнкция, как в случае прямой системы.

Условие остановки то же самое: факт является конъюнкцией литералов, согласованный граф решения заканчивается в фактах. Мы можем наращивать исходную цель, представленную в форме И/ИЛИ, с помощью правил, до тех пор, пока она не обопрется на факты.

Рассмотрим **пример**. Построим обратную систему дедукции на основе правил, которая будет решать неравенства.

П1: $x > 0 \;\&\; y > 0 \rightarrow xy > 0$

П2: $x > 0 \;\&\; y > z \rightarrow x+y > z$

П3: $x > w \;\&\; y > z \rightarrow x+y > w+z$

П4: $x > 0 \;\&\; y > z \rightarrow xy > xz$

П5: $1 > w \;\&\; x > 0 \rightarrow x > xw$

П6: $x > (wz+yz) \rightarrow x > (w+y)z$

П7: $x > wy \;\&\; y > 0 \rightarrow x/y > w$

Имеется семь правил. В каждом из них в правой части стабильно стоит один литерал, поскольку там есть только один предикат. Слева может быть один предикат или конъюнкция предикатов. Таким образом, эти правила подходят для применения в обратной системе дедукции.

Имеется конъюнкция фактов, заключающаяся в том, что

Факты: A, B, C, D > 0 C > D

Целью является неочевидное утверждение:

Цель: B(A+C)/D > B

Ход рассуждений представлен на рис. 34.

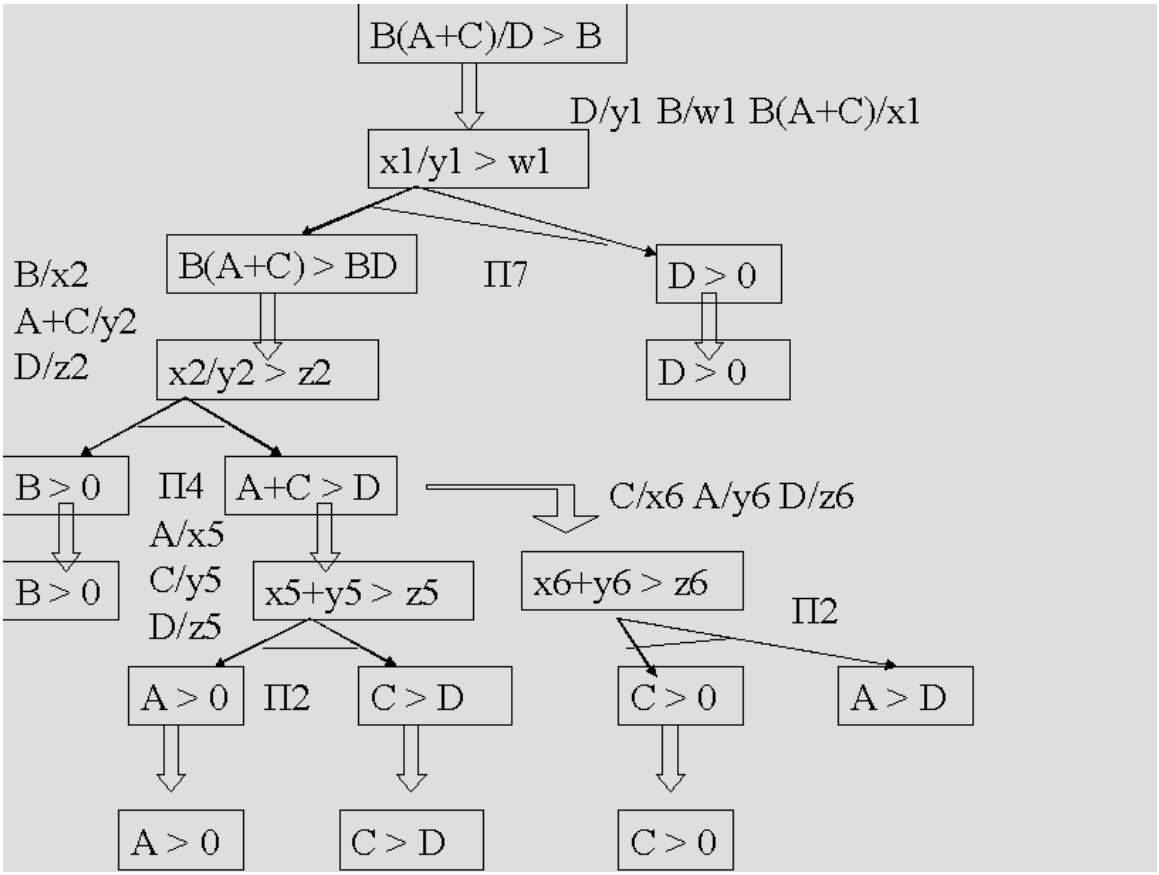


Рис. 34. Решение неравенства с помощью обратной системы дедукции.

Давайте посмотрим на первое неравенство. Здесь главная операция – деление. При унификации, очевидно, сработает правило 7, других правил с делением в правой части нет. Это первое подходящее правило. Правило 7 имеет вид

$$x > wy \;\&\; y > 0 \rightarrow x/y > w,$$

где x отождествиться с B(A+C), при этом y отождествится с D; w отождествится с B. Подставляем правую часть правила. Получаем две новых цели: D > 0 (это дано) и B(A+C) > BD. Здесь будет применено первое правило, в котором есть умножение в правой части. Появляются два подцели: B > 0 и A+C > D. Ко второй части применимы два правила – четвертое и второе. Но мы по-разному можем провести унификацию ввиду свойства коммутативности. Первая подстановка ведет к результату, вторая заводит в тупик, и с этим ничего не сделать. Проверяем, что все подстановки согласованы, значит все должно быть в порядке. Обратите внимание, что каждый экземпляр правил – со своими личными переменными, чтобы случайно не оказалось одинаковыми переменные под разными кванторами.

5.5. Комбинация прямой и обратной систем

Мы рассмотрели один ограниченный случай (прямая система) и другой ограниченный случай (обратная система). Возникает естественная идея, объединить эти два ограниченных случая в надежде получить неограниченный.

5.5.1. Правило гашения

В результате объединения прямой и обратной систем дедукции на основе правил получается следующее.

Имеются факты, представленные в форме И/ИЛИ, имеются цели, представленные в форме И/ИЛИ, и имеются два сорта правил: прямые правила в форме L→W, обратные правила в форме W→L. Они будут навстречу друг другу наращивать графы решения, пока эти графы не столкнуться. Другими словами, мы вводим три вида резолюции:

- резолюция фактов и прямых правил, которая является частным случаем метода резолюций Робинсона, которую мы уже обсудили;
- резолюция целей и обратных правил, которая является частным случаем двойственного метода резолюций;
- резолюция фактов и целей.

Резолюция фактов и целей называется правилом гашения.

Правило гашения заключается в следующем: две вершины гасят друг друга, если они либо унифицируются, либо гасятся исходящие их них *k*-связки.

Условие окончания выглядит в этом случае так: корень фактов гасит корень целей и подстановки на дугах соответствия согласованы.

Рассмотрим позитивный **пример**, чтобы понять преимущества комбинированный прямой и обратной системы (это пример уже рассматривался в разд. 4.2.5).

Факт:	существуют умные студенты
Обратное правило:	программисты знают метод резолюций
Прямое правило:	студенты не знают метода резолюций
Цель:	существуют умные не программисты

Мы сами решаем, какое правило прямое, а какое правило обратное. Это накладывает дополнительную ответственность на того, кто заносит эту информацию, но с другой стороны позволяет то, что не позволяет метод резолюций: сохранить импликативность, показать, в какую сторону надо применять рассуждение.

В результате решение находится достаточно легко (рис. 35).

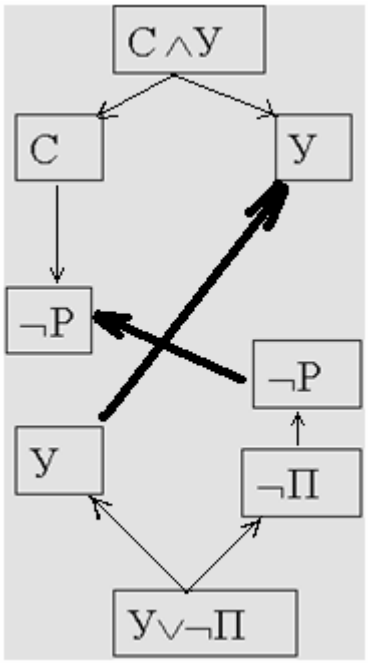


Рис. 35. Комбинация прямой и обратной систем дедукции.

Одно применение правил для наращивания фактов, одно применения правил для наращивания целей – и сразу же гашение. Другими словами, в отличие от метода резолюций, эта система сработает за три шага, выполняя всего три унификации. При этом ей не надо думать, какие правила применять, не потребовалось никакой стратегии. Все применимы правила былыи применены и в результате получено решение.

5.5.2. Неполнота гашения

К сожалению, в этом достаточно эффектном примере есть сложность. Правило резолюции логично и полно, и правило применения прямых правил, как частный случай правила резолюции логично и полно, и правило применения обратных правил, как частный случай двойственного метода резолюции, логично и полно. Однако с правилом гашения все обстоит не так просто и хорошо. Оно не универсально, т.е. оно логично, но не полно. Существуют такие выводы, которые не могут быть получены по правилу гашения (рис. 33).

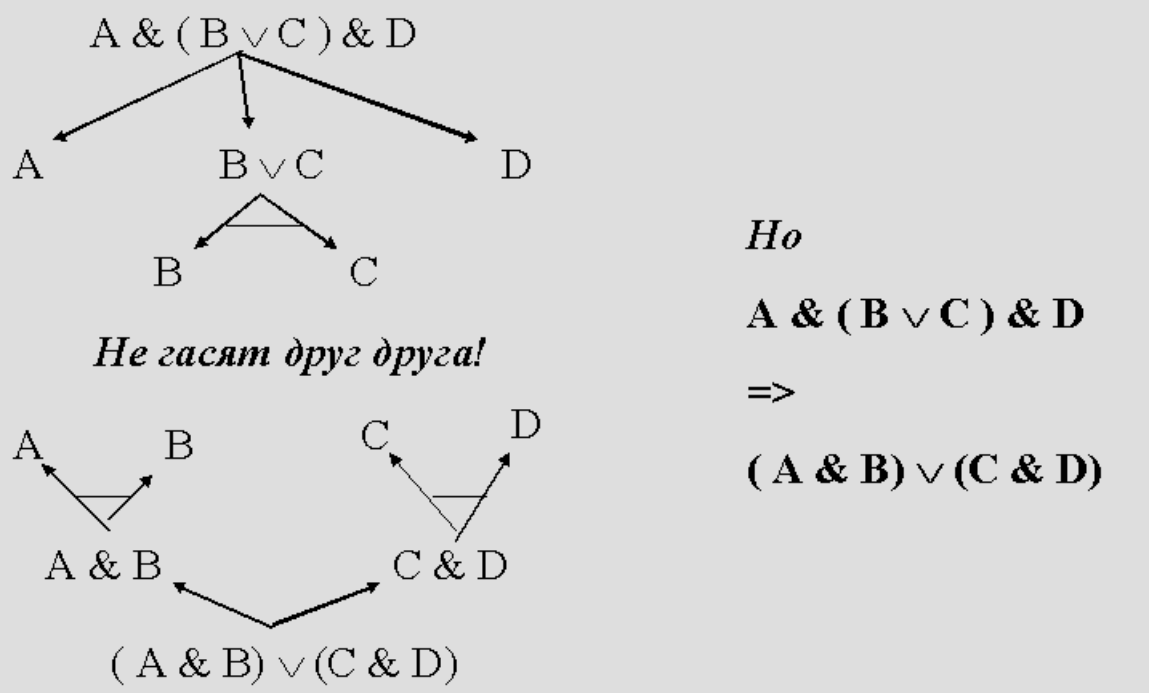


Рис. 36. Неполнота правила гашения.

Нижняя формула является логическим следствием верхней формулы

$$(A \,\&\, B) \vee (C \,\&\, D) \Rightarrow A \,\&\, (B \vee C) \,\&\, D.$$

Более того, очень хорошо вырастают деревья навстречу друг другу, очень хочется их погасить – буквы одни и те же в обоих случаях. Но этого сделать нельзя, поскольку индуктивное определение следующее: либо унифицируемые, либо гасятся их *k*-связки. А в данном случае *k*-связки как раз и не гасятся. Это разные структуры по количеству дуг: наверху три графа решения, а внизу два.

В данном случае комбинированная система дедукции не сработала, и причина состоит в том, что само представление И/ИЛИ не носит всеобщего характера. Все же что-то мы теряем, когда общую формулу исчисления предикатов переводим в форму И/ИЛИ. Такого замечательного факта, который имеет место в методе резолюций, что после сведения к предложениям, объединение исходных посылок и отрицание теоремы не выполнимо тогда и только тогда, когда целевая теорема следует из посылок, здесь, к сожалению, нет. То есть, может быть такая ситуация, что целевая теорема следует из посылок, а построенное множество не хочет опровергаться.

Остается открытым вопрос – можно ли подправить форму И/ИЛИ таким образом, чтобы резолюция фактов и целей сделалась логичной и полной так же, как резолюция фактов и правил и резолюция правил и целей.

5.5.3. Метазнания в системах дедукции

Системы дедукции на основе правил хорошо подходят для представления фактов, правил и целей и дают почти полную свободу при решении вопроса, что считать фактами, что правилами, а что целями.

Однако есть управляющие знания (метазнания), очень важные для ПСИИ, которые трудно вместить в системы дедукции на основе правил. Мы приводим три

характерных примера метазнаний. Часть их них неявно была применена в примерах ранее.

1. Учет потенциальной ширины ветвления.

Допустим, что в системе дедукции есть группа правил классификации. Например

собака(x) → животное(x)

кошка(x) → животное(x)

и так далее, таких правил может быть много. Формально, такие правила можно применять как в прямом, так и в обратном направлении. В каком направлении более целесообразно? Это зависит от предметной области, от типа решаемых системой задач. Знанием о целесообразном направлении применения правил обладает эксперт, который строит систему дедукции. Например, эксперт знает, что правила классификации не следует применять как обратные правила. Дело в том, что применение таких правил в обратном направлении даст очень большую ширину ветвления при разрастании графа целей. Такие правила нужно применять в прямом направлении.

Если эксперт считает, что правила классификации в данной системе дедукции все же нужно применять в обратном направлении, то ему следует переписать их в контрапозитивной форме:

¬животное (x) → ¬собака (x)

¬животное (x) → ¬кошка (x)

и так далее.

2. Порядок применения правил.

Допустим, что в системе дедукции есть прямые правила вида

$$P \rightarrow (Q_1 \vee \ldots \vee Q_k)$$

или обратные правила вида

$$(Q_1 \,\&\, \ldots \,\&\, Q_k) \rightarrow P$$

Применение таких правил наращивает граф k-дугами. Большое значение имеет порядок, в котором далее будут рассматриваться узлы Q_1 & … & Q_k. Например, в обратной системе дедукции на основе правил для решения неравенств (см. разд. 5.4.2, рис. 34) после применения правила 4 первым нужно рассматривать выражение B > 0, а не выражение A + C > D. Дело в том, что это выражение *уже* имеет форму факта. Эксперт понимает, что если нам необходим для доказательства какой-то конкретный факт, то возможны два случая: либо этот факт дан в условиях задачи, либо нет. Из пальца факт не высосешь. Если факт дан (как в примере 5.4.2), то следует продолжать рассматривать другие выражения, в надежде свести их к данным фактам. Если же факта нет, он не подтверждается, то нужно прекратить рассмотрение других выражений, так как рассмотрение ничего не даст: данная k-связка все равно не войдет в граф решения.

Такого рода информация является метазнаниями, которые вносятся в дедуктивные системы с помощью специальных приемов. Например, с помощью оценочных функций, как в продукционных системах.

3. Условное применение правил.

Рассмотрим еще раз пример с решением неравенств (см. разд. 5.4.2, рис. 34). Из соображений транзитивности хотелось бы сказать, что факт C > O – лишний. Возникает соблазн вставить восьмое правило: $x > y$ & $y > z \rightarrow x > z$. Очень опасное правило! Дело в том, что его правая часть применима всегда, и она унифицируема с

чем угодно в этой системе. Здесь две простые переменные связаны единственным предикатом, который есть в системе. Какой бы узел ни стоял в графе И/ИЛИ, его всегда можно унифицировать с правой частью правила транзитивности. А это означает, что такое правило будет использоваться постоянно. Можно сказать, что правило транзитивности слишком применимо! Такие правила очень опасно использовать. Их можно спрятать в унификацию (так обычно и поступают), либо – (очень красивое решение) добавить в наш алгоритм метазнание, говорящее о том, что есть правила, которые можно применять всегда, когда они применимы, а есть правила, которые применяются только при особых условиях. Это некие условия о структуре текущей базы знаний, которая управляет применением или неприменением правил. Метазнание для транзитивности такое: правило транзитивности можно применять тогда и только тогда, когда один из конъюнктов в левой части является фактом. Если это не так, то в результате применения правила транзитивности размножаются цели. Если же это так (если один из конъюнктов, стоящих в левой части правила транзитивности, сопоставим с фактом), то в результате происходит просто передвижение по фактам. При этом дерево решения остается линейным. Дерево не будет ветвиться, а будет просто виться по цепочке транзитивности, что и требуется от применения правила транзитивности. Применяя его просто так, порождается y, с которым неизвестно что делать. Можно согласиться на порождение этого y, только если удается взамен избавиться от x.

Вывод из приведенных примеров метазнаний состоит в следующем. Метазнания – это не знания о предметной области, для которой строится система дедукции. Это знания о знаниях, от том, как применять и использовать знания о предметной области, какими свойствами обладают знания о предметной области и т.д.

Очень хорошие результаты дает двухуровневая система дедукции на основе правил. Такая система на верхнем уровне содержит метазнания, также представленные, например, в форме правил. Фактами в системе верхнего уровня является описание текущего состояния системы нижнего уровня. Целью является построение (синтез) стратегии управления для системы нижнего уровня. На втором уровне находится обычная система дедукции, правила, факты и цели которой берутся из предметной области. Система верхнего уровня определяет для системы нижнего уровня в каком направлении нужно применять правила, в каком порядке рассматривать узлы, при каких условиях можно применять правила и т.д.

Тема 6. Автоматический синтез программ

Прикладные системы с элементами искусственного интеллекта – это передовой край информатики. Программирование само по себе – основа автоматизации какой-либо деятельности. Естественно, что автоматизация самого программирования всегда являлась самой престижной задачей для программистов.
Вообще говоря, идеальная картина выглядит следующим образом (рис. 37):

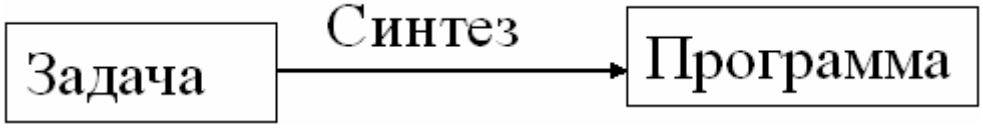


Рис. 37. Идеал автоматического программирования.

Действительно, насколько было хорошо, если бы по постановке задачи, автоматически, не тратя время и силы на программирование и отладку можно было бы получить готовую программу для решения данной задачи. В этой теме рассматривается вопрос о том, что действительно можно получить уже сейчас, а что пока еще нельзя.

6.1. Задача автоматического синтеза программ

Автоматический синтез программы проводится непосредственно по спецификации задачи. Весь вопрос в том, как специфицирована задача, на каком языке дано описание задачи.
Рассмотрим обычный язык программирования – Паскаль, Си или нечто подобное. Компьютер непосредственно не выполняет текст программы на языке программирования. Компьютер выполняет только машинные команды из той системы команд, которую умеет выполнять процессор. Так что текст программы на языке программирования – это тоже спецификация задачи, а транслятор – это тоже синтезатор. Раньше, когда техника трансляции была еще недостаточно развита, трансляторы иногда относили к области, пограничной с ИИ, и говорили о системах автоматического программирования, если текст программы можно было писать не в машинном коде, а на языке более высокого уровня. Но в настоящее время трансляция не считается автоматическим синтезом программ. Сейчас говорят непроцедурная спецификация, спецификация высшего уровня и т.д., подчеркивая, что задача должна быть описана какими-то средствами, непохожими на обычный язык программирования.

6.1.1. Классификация подходов к синтезу программ

На данный момент существует четыре подхода к автоматическому синтезу программ.
1. *Дедуктивный синтез* (логический вывод). В этом случае спецификация задается теоремой в формальном исчислении; программа извлекается из доказательства теоремы. В этом учебнике рассматривается дедуктивный синтез с использованием метода резолюций.
2. *Индуктивный синтез* (синтез программ по примерам). В этом случае спецификация задается набором примеров выполнения программы; программа извлекается из примеров путем обобщения, выявления последовательностей,

прогрессий и т.п. В этом учебнике рассматривается индуктивный синтез циклических программ, работающих с целочисленными массивами.

3. *Трансформационный синтез* (трансляция). Этот метод заключается в поэтапном преобразовании исходной спецификации в исполнимый код. Исходная спецификация задается на языке более высокого уровня, нежели обычный язык программирования. Используются графические средства, такие, как унифицированный язык моделирования UML, различные математические объекты, такие как системы дифференциальных уравнений и тому подобное. В спецификации используются непосредственно понятия предметной области, без указания на способ их представления в программе. Спецификации для таких систем часто называют непроцедурными, декларативными и т.п., чтобы подчеркнуть их отличие от привычных императивных языков программирования. Как правило, системы трансформационного синтеза очень сильно привязаны к конкретной предметной области. Экспертные знания из конкретной предметной области являются решающими. В этом учебнике системы трансформационного синтеза не рассматриваются.

4. *Утилитарный синтез* – частные приемы в частных случаях (генераторы отчетов и прочее); по сути все, что не попадает в первые три пункта. Здесь рассматривается структурный синтез программ – простой, но очень эффектный метод, который позволяет синтезировать программы ограниченного вида в узком классе предметных областей. Зато этот метод в своей области настолько эффективен, что используется в промышленных масштабах.

6.1.2. История развития синтеза программ

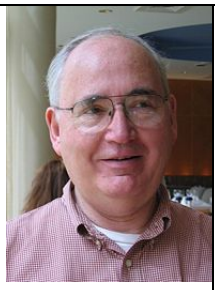
Начало автоматическому синтезу программ было положено в 60-е годы XX века и первоначально синтез программ не отделялся от обычной трансляции языков программирования.

С появлением метода резолюций дедуктивный синтез программ резко выделился из общего потока работ по автоматизации программирования за счет использования аппарата математической логики.

В 70-е годы наблюдался повышенный интерес к этой тематике, были получены значительные результаты. Появилась аксиоматика Хоара, деривационная семантика Скотта. Параллельно было выявлено несколько важнейших фактов. Было установлено, что формальная логическая верификация (автоматическое доказательство правильности программы) эквивалентна задаче дедуктивного синтеза и в общем случае (без знания инвариантов циклов) алгоритмически неразрешима.

Сэр ***Чарльз Энтони Ричард Хоар*** *(Charles Antony Richard Hoare), род. 1934 — британский учёный, специализирующийся в области информатики и вычислительной техники. Наиболее известен как разработчик алгоритма «быстрой сортировки» (1960), на сегодняшний день являющегося наиболее популярным алгоритмом сортировки.*
Другие известные результаты его работы: язык Z спецификаций и параллельная модель взаимодействия последовательных процессов (CSP, Communicating Sequential Process). В числе его заслуг — разработка логики Хоара, научной основы для конструирования корректных программ, используемой для определения и разработки языков программирования.
В 2000 он был удостоен рыцарского титула за заслуги в области образования и компьютерных наук.

Дана Стюарт Скотт *(Dana Stewart Scott), род. 1932 — американский учёный в области математики и информатики.*
Исследования Скотта связанны с теорией моделей, теорией автоматов, модальной и интуиционистской логиками, конструктивной математикой и связью между логикой и теорией категорий. Философские интересы лежат в области оснований логики, философии математики и семантического анализа естественных языков. Работы в области информатики направлены на разработку денотационной семантики языков программирования и математических основ вычислимости.



Затем, в 80-е годы XX века наблюдался некоторый спад интереса к этой тематике, и в то же время работы велись, и были получены исчерпывающие ответы на некоторые вопросы синтеза программ. В частности, Н.Н. Непейвода и другие исследователи показали, что из интуиционистского доказательства теоремы существования определенного вида автоматически и эффективно извлекается программа на обычном императивном языке программирования. Тем самым задача автоматического синтеза программ была сведена к задаче автоматического доказательства теорем.

Непейвода Николай Николаевич *(род. 1949) — математик, логик, философ, информатик. Является одним из основателей теории неформализуемых понятий и теории логического синтеза программ на базе конструктивных логик, создал две из четырёх известных классов конструктивных логик, разработал методику логического подхода, ставшего естественной альтернативой системному подходу.*

В 90-е годы XX века наблюдался застой. По мнению автора, это было связано с тем, что доминирование объектно-ориентированного программирования не способствовало автоматическому синтезу программ.
В настоящее время наблюдается вторая волна интереса к автоматическому синтезу, что проявляется в появлении новых (часто хорошо забытых старых) подходов, таких как генеративное программирование, исполнимые визуальные спецификации, модельно-центрированная разработка (Model Driven Architecture – MDA) и другие. Видимо, в ближайшее время следует ожидать появления новых важных результатов в этом направлении.

6.2. Дедуктивный синтез программ

Дедуктивный синтез программ – наиболее развитый в настоящее время вид автоматического синтеза программ.

6.2.1. Схема дедуктивного синтеза

Общая схема дедуктивного синтеза следующая:

Задача → Теорема → Доказательство → Программа

Как уже сказано в предыдущем разделе, вопрос о том, как реализовать последнюю стрелку в этой схеме, полностью решен в настоящее время.
Вопрос о том, как реализовать первую стрелку в этой схеме, то есть, как перейти от неформальной задачи к ее формальной спецификации в форме теоремы, выходит за рамки данного учебника. Это философский, гносеологический вопрос.
Таким образом задача дедуктивного синтеза ставится следующим образом. Заданы два предиката:
$P(x)$, который называется *предусловие* и
$Q(x)$, который называется *постусловие*.
Требуется автоматически доказать (построить вывод из аксиом) теорему существования следующего вида:

$$\forall x\,(P(x) \to \exists y\, Q(x, y)).$$

Смысл этой теоремы можно прочитать так: для любых входных данных x, если входные данные удовлетворяют предусловию P, то существуют выходные данные y, такие что удовлетворяется постусловие Q.
Заметим, что после сколемизации исходная формула приобретает вид

$$\forall x\,(P(x) \to Q(x, f(x)))$$

Найти явное выражение для этой неизвестной, введенной при сколемизации функции f и есть цель автоматического дедуктивного синтеза программ.
Обычно, используя нотацию Хоара, это записывают так

$$P(x)\ \{y := f(x)\}\ Q(x, y)$$

Здесь в середине стоит оператор присваивания, символизирующий выполнение синтезированной программы, слева от него предусловие, а справа – постусловие. Прочитать эту запись можно следующим образом: если до начала выполнения программы выполнено предусловие для входных данных, то после выполнения присваивания (то есть после вычисления выходных данных по входным данным с помощью синтезированной функции f), будет выполнено постусловие.

6.2.2. Тотальная корректность

Когда говорят, что хотят синтезировать программу, неявно подразумевают, что хотят синтезировать тотально корректную программу, которая обладала бы тем свойством, что если предусловие выполнено, то постусловие тоже выполнено. Эта постановка задачи, если ее оставить без уточнения, не очень-то разумна.

Пример.

Ниже приведена программа, (якобы) решающая поставленную задачу:

```
proc f(x);
     y:=x;
     while ¬Q(x,y) do y:=random(y)end while
     return y
end proc
```

Это «универсальный» алгоритм синтеза тотально корректных программ. Данная программа решает поставленную задачу. Возможно, она не очень быстро закончит работу ☺, но уж точно, когда закончит – обязательно будет выполнено постусловие. Но это не совсем то, чего мы добиваемся. Разумеется, этот «универсальный алгоритм» не более чем шутка, но в этой шутке содержится намек на то, что когда мы рассуждаем про пред- и постусловия, надо ясно понимать, что мы хотим получить не просто тотально корректный ответ. Мы хотим получить по возможности максимально эффективную реализацию функции f. И сделать это тоже хотим по возможности максимально эффективно.
Наверное, стоит поставить какие-то ограничения сверху на эффективность функции, которую мы хотим получить, и от этого появляется ограничение снизу на эффективность алгоритма, с помощью которого можно синтезировать разумную программу.
Нас не интересуют алгоритмы, которые работают бесконечно долго. Мы хотим получить разумно работающий алгоритм f. Но разумно работающий алгоритм нельзя получить даром. Мы должны быть готовы заплатить разумную цену.

6.2.3. Реализуемость

Во втором примере обсуждается вопрос реализуемости, который часто пропускают при постановке задач, особенно в генеративном программировании.

Допустим, есть эвристический алгоритм синтеза, который в некоторых случаях находит правильную функцию *f*, а в некоторых, возможно, находит неправильную или вовсе ничего не находит. Но зато он эффективно работает. Возникает сильный соблазн «починить» этот алгоритм:

```
proc f1(x);
     y:=f(x);
     if Q(x,y) then return y
     else return fail
     end if
end proc
```

Допустим, алгоритм выдает функцию *f*, которую можно «завернуть» в функцию *f*1, которая точно будет не тотальна, но, по крайней мере, корректна. Реализуемая ли эта программа? Сможет ли она вычислить, например, предикат Q, заданный следующим образом:

$$Q(x, y, z) = \forall n>2\ \exists x, y, z\ (x^n+y^n=z^n)$$

Не всегда пред- и постусловия таковы, что их вычисление, т.е. прямое включение пред- и постусловий внутрь кода программы допустимо.

Мы должны ясно и точно заранее решить, чему мы можем позволить появиться в тексте синтезированной программы, а что там появляться не должно, хотя, может быть и содержится в спецификации задачи.

6.3. Дедуктивный синтез на основе метода резолюций

Рассмотрим серию конкретных примеров различных методов автоматического синтеза программ с использованием метода резолюций.

6.3.1. Синтез программы в функциональной форме

Самый простой пример: синтез программы уровня языка ассемблера в функциональной форме.

Пусть имеется одноадресная машина с сумматором и с тремя ячейками, которые адресуются идентификаторами a, b и c. Пусть в это машине есть команды:

load – загрузить число из ячейки в сумматор,

add – прибавить содержимое ячейки к содержимому сумматора,

store – записать содержимое сумматора в ячейку.

Мы хотим синтезировать программу для решения задачи c := a + b.

Введем только один предикат $P(u, x, y, z, s)$, который означает следующее: в сумматоре находится значение u, в первой ячейке – x, во второй – y, в третьей – z, машина находится в состоянии s. Тогда аксиоматика и вывод по методу резолюций с использованием предиката ANS имеют следующий вид.

1. $\neg P(u, x, y, z, s1) \vee P(x, x, y, z, \text{load}(a, s1))$
2. $\neg P(u, x, y, z, s2) \vee P(u+y, x, y, z, \text{add}(b, s2))$
3. $\neg P(u, x, y, z, s3) \vee P(u, x, y, u, \text{store}(c, s3))$

4. $P(e1, e2, e3, e4, d)$
5. $\neg P(u, x, y, x+y, s) \vee \text{ANS}(s)$
6. $\neg P(x+y, x, y, z, s1) \vee \text{ANS}(\text{store}(c, s))$ 5,3
7. $\neg P(x, x, y, z, s2) \vee \text{ANS}(\text{store}(c, \text{add}(b, s)))$ 6,2
8. $\neg P(u, x, y, z, s3) \vee \text{ANS}(\text{store}(c, \text{add}(b, \text{load}(a, s))))$ 7,1
9. $\text{ANS}(\text{store}(c, \text{add}(b, \text{load}(a, d))))$ 8,4

Первая строчка – описание семантики команды «загрузить содержимое ячейки *a* в сумматор». Аналогично описываются команды add и store (строки 2 и 3).

Аксиома, которая здесь используется (строка 4), такова: машина всегда находится в некотором состоянии, и в ячейках и в сумматоре что-то есть.

Цель: мы хотим, чтобы в ячейках *a* и *b* находились какие-то числа *x* и *y*, а в ячейке *c* – число *x*+*y*. То состояние, в котором это будет достигнуто, и есть интересующий нас ответ. Здесь применяется обычный метод резолюций. Резольвируем предложения 5 и 3 по P и так далее. В результате получается программа в функциональной форме на ассемблере, которая решает поставленную задачу. Чтобы вычислить сумму того, что находится в первых двух ячейках, нужно сначала загрузить первую ячейку, потом прибавить вторую, а результат отправить в третью ячейку.

Сразу видно, что данный пример на практике малопригоден, поскольку модель слишком упрощена. В реальной машине и регистров больше, и ячеек много, и команд много, и резольвироваться они будут очень легко. Поэтому метод резолюций будет резольвировать, резольвировать, а появления простой программы мы, может быть, так и не дождемся.

Еще одно замечание по этому примеру. Программу из трех машинных команд мы специфицировали и синтезировали с помощью 8 предложений, содержащих 16 литералов, 6 связок и 7 термов. Грубо говоря, ассемблер оказался примерно в 10 раз лаконичнее языка исчисления предикатов. А ведь известно, что ассемблер далеко не самый лаконичный язык! Современные языки программирования еще в 10 раз лаконичнее ассемблера (на одну операцию языка высокого уровня при трансляции порождается в среднем около 10 машинных команд). Получается, что формально специфицировать и автоматически синтезировать такую простую программу оказывается примерно в 100 раз дороже, чем просто написать эту программу вручную.

Если еще учесть, что среднего школьника можно обучить ассемблеру примерно за месяц, а научить читать, писать и понимать язык исчисления предикатов среднего студента в университете можно примерно за год, то получается, что автоматическое программирование в стиле разобранного примера в 1000 более дорогое удовольствие, чем традиционное ручное программирование.

Следовательно, должны быть серьезные причины для применения методов автоматического синтеза программ.

В определенных ситуациях такие причины существуют. Например, автоматически синтезированная программа правильна по построению. Ее не нужно отлаживать и тестировать. Можно сказать, что синтезированная программа извлечена из доказательства своей правильности. Бывают такие специальные случаи программирования, когда никаких затрат не жалко, если можно гарантировать на 100%, что программа правильна. Тестирование не может гарантировать

правильность на 100%. Кроме того, иногда тестирование оказывается дороже формального доказательства правильности или натурное тестирование принципиально затруднено. Например, системы управления оружием и боевые роботы.

6.3.2. Синтез блок-схемы

Давайте улучшим имеющийся алгоритм. В только что описанном примере мы получили программу в функциональной форме. Хочется, тем не менее, получить ту же самую программу в более привычном для нас виде.

Посмотрим, как из доказательства теоремы существования, полученной методом резолюций, может быть извлечена более привычная форма программа – блок-схема.

Положим, что у нас имеется доказательство, представленное в древовидной форме, то есть не в линейной записи, а в виде дерева, где корнем дерева является наше целевое утверждение, которое мы получили в результате работы.

Тогда следующий алгоритм преобразует дерево доказательства в исполнимую блок-схему программы.

Вход: дерево вывода T_0 (в узлах – предложения).

Выход: схема программы T (дерево решений).

Если имеется такое дерево, применим к нему следующие шаги преобразования.

Шаг 1: поместить на дуги отрицание литерала, по которому проводилась резолюция вместе с унификатором (получим дерево T_1).

Шаг 2: удалить все узлы (и инцидентные им дуги), где нет предиката ANS (получим дерево T_2).

Шаг 3: перевернуть дерево корнем вверх и удалить предложения из узлов (получим дерево T_3).

Шаг 4: удалить литералы на дугах, выходящих из узлов с одним выходом. При этом унификатор остается (получим дерево T).

Пример: синтез программы, которая вычисляет знак числа:

```
y := if x>0 then 1 else -1 end if.
```

Обозначим через *P* предикат, который проверяет, что его аргумент больше нуля: $P(x) := x > 0$, а предикат $Q(x, y)$ – постусловие, которое связывает входные данные *x* с выходными данными *y*.

Тогда исходная аксиоматика, то есть спецификация задачи? имеет следующий вид:

$$P(x) \to Q(x, 1) \;\&\; \neg P(x) \to Q(x, -1)$$

Преобразуем в предложения и проведем вывод методом резолюций.

1. $\neg P(x) \vee Q(x, 1)$
2. $P(x) \vee Q(x, -1)$
3. $\neg Q(x, y) \vee \text{ANS}(y)$
4. $\neg P(x) \vee \text{ANS}(1)$ 1,3
5. $P(x) \vee \text{ANS}(-1)$ 2,3
6. $\text{ANS}(1) \vee \text{ANS}(-1)$ 4,5

Аксиоматика ясна, первое и второе – это исходные положения, третье – это вопрос, который записан с участием предиката ANS.

Проводим обычное резольвирование. Резольвируем 1 и 3 предложение по предикату *Q*, получает $\neg Q(x, y) \vee \text{ANS}(1)$. Дальше резольвируем 2 и 3 по предикату

Q еще раз и получаем $P(x) \vee$ ANS(–). И резольвируем два последних предложения по предикату *P*. В результате получаем ANS(1) $\vee$ ANS(–). Доказано, что ответ существует (рис. 38).

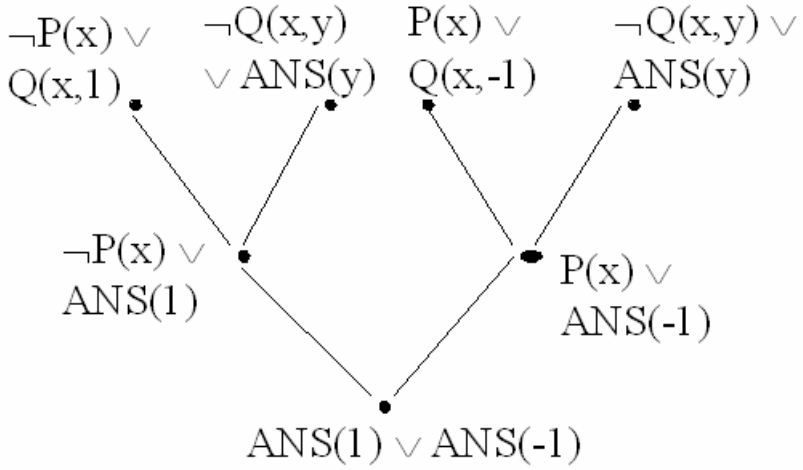


Рис. 38. **Дерево доказательства.**

На верху написаны исходные предложения, ниже, в узлах дерева – результаты их резольвирования, и еще ниже – результаты последнего резольвирования.

Это логическая часть. Но из этого дерева легко извлекается реальная блок схема.

Первый шаг алгоритма состоит в том, что мы навешиваем на это дерево подстановки и отрицание тех литералов, по которым проводилось резольвирование. На втором шаге мы убираем лишнюю информацию из этого дерева, то есть, убираем все узлы, в которых нет ANS. Это два узла на самом верхнем уровне дерева. Третий шаг – мы переворачиваем дерево и убираем надписи предикатов на тех узах, в которых только одна исходящая дуга. Результат представлен на рис. 39.

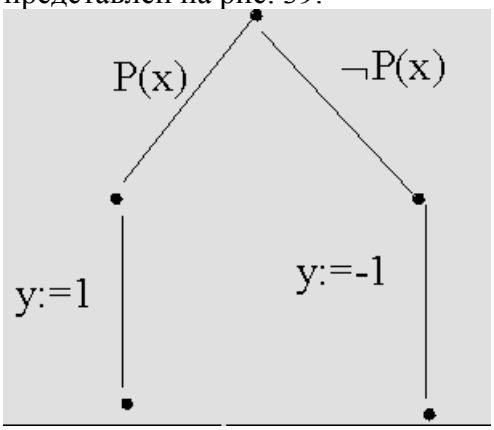


Рис. 39. Синтезированная блок-схема.

Эта блок-схема решает задачу, и получается из доказательства автоматически.

6.3.3. Синтез невыполнимой программы

Однако в предыдущем примере есть подводный камень.

Возьмем ту же самую аксиоматику, но резольвирование проведем по-другому.

1. $\neg P(x) \vee Q(x, 1)$

2. $P(x) \vee Q(x, -1)$
3. $\neg Q(x, y) \vee \text{ANS}(y)$
4. $Q(x, 1) \vee \text{Q}(x, -1)$ 1,2
5. $\text{Q}(x, -1) \vee \text{ANS}(1)$ 3,4
6. $\text{ANS}(1) \vee \text{ANS}(-1)$ 3,5

Сначала резольвируем 1 и 2 предложения по P, получаем $Q(x, 1) \vee Q(x, -1)$. Потом убираем Q, резольвируя его с 3 и 4 предложениями. Получаем тот же самый ответ $\text{ANS}(1) \vee \text{ANS}(-1)$. Теорема доказана. Дерево, соответствующее новому доказательству представлено на рис. 40.

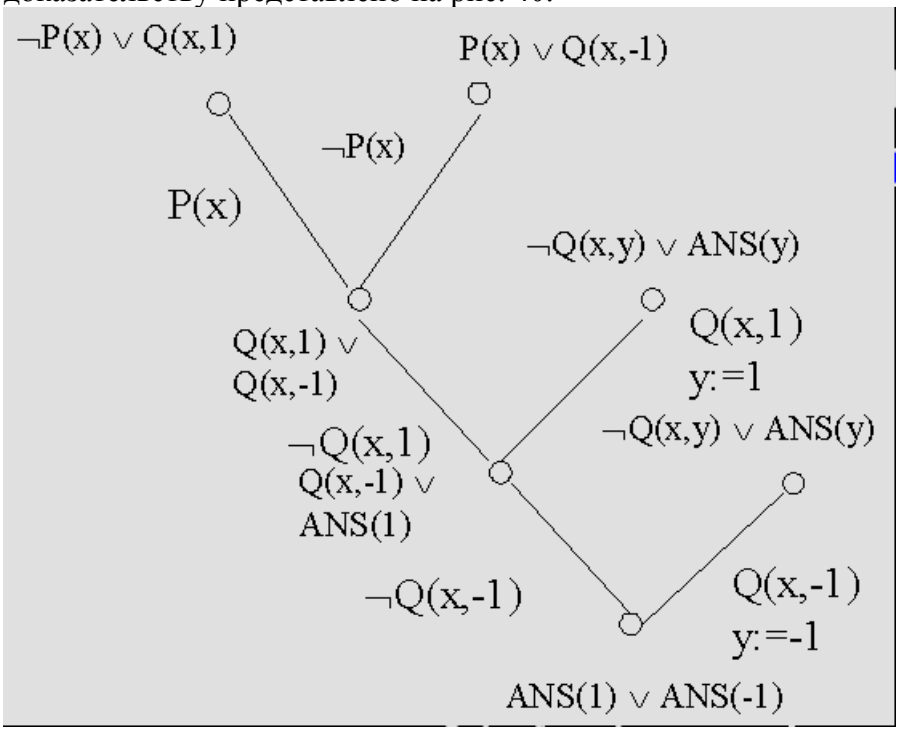


Рис. 40. Другое дерево доказательства.

Это дерево имеет немного другую структуру. Однако если к нему применить только что приведенный алгоритм извлечения программы из доказательства, то получится следующая программа (рис. 41):

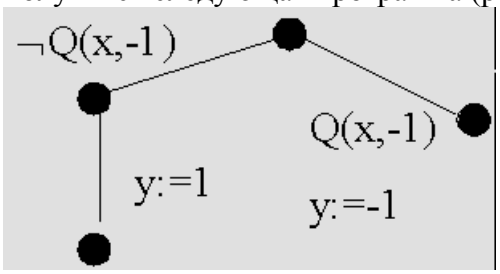


Рис. 41. Невыполнимая блок-схема.

Если не верно, что ответом должен быть –1, то ответ 1. А если ответом должен быть –1, то ответ –1. Это не совсем такая программа, которую хотелось бы видеть в ответе. Это конкретный пример нереализуемости, говорящий о том, что надо быть очень осторожными относительно того, может или не может нечто появиться в программе. Не все, что легко написать внутрь программы, на самом деле реализуемо, то есть вычислимо. В данном случае предикат Q не реализуем,

программа бессмысленна. Алгоритм не сработал. Попробуем «починить» его с помощью теории примитивной резолюции.

6.3.4. Примитивная резолюция

Символы исходной спецификации делятся на два сорта – те, которым разрешается попадать в программу, и те, которым не разрешается попадать в программу.

Символы исходной спецификации (константы, функциональные символы, предикаты), которым разрешается появляться в синтезированной программе, называются *примитивными.*

В нашей постановке задачи предикат P и константы 1 и –1 являются примитивными, и им разрешено появляться в программе. Предикат Q – непримитивный, и ему не разрешается попадать в программу.

Предложение, содержащее предикат ANS, называется *жизненным*.

После преобразования доказательства в блок-схему, те узлы, где был ANS останутся, а где этого предиката не было – не останутся.

Переменная, входящая в жизненное предложение, называется *жизненной.*

Те переменные, которые входят в такое предложение, останутся в виде переменных программы, а те, которые не входят – они не выживут. Они не нужны и являются лишними.

$\neg P(x) \vee Q(x, 1)$ – не жизненное предложение;

$Q(x, -1) \vee \text{ANS}(1)$ – жизненное предложение.

Примитивная резолюция – это тот же самый метод резолюций, на который наложено два дополнительных условия. Разрешается резольвировать в двух случаях.

1. Если оба резольвируемых предложения C_1 и C_2 нежизненные, или одно из них жизненное, а второе нет, то в этом случае все константы и функции, которые подставляются вместо жизненных переменных в φ, являются примитивными.

$$C1 \vee L1 \quad C2 \vee \neg L2 \rightarrow C\varphi, \text{ где } \varphi = \text{Н.О.У.}(L1, L2)$$

Первое условие определяет фактически, какие присваивания мы разрешаем себе синтезировать. Синтез присваиваний, в правых частях которых используются примитивные функции и константы допускается допускается.

2. Если оба резольвируемых предложения C_1 и C_2 жизненны (а это означает, что у нас в блок-схеме появятся ветвления), то требуется не только, чтобы все константы и функции, участвующие в подстановках в φ, были примитивными, но и чтобы предикаты L_1 и L_2 тоже были примитивными.

Теорема. (без доказательства)

Примитивная резолюция полна.

То есть если существует какое-то доказательство, то существует и примитивное доказательство.

Пример.

Обратимся еще раз к рис. 40. В верхушке дерева имеется два нежизненных предложения, и подстановка допустима. Второй шаг – одно жизненное предложение, а другое нет; резольвирование ведется по Q. Но подстановка тоже допустима. Третий шаг – оба предложения жизненные, а резольвирование ведется по предикату Q, который не является примитивным. Другими словами, третий

вывод не является примитивным и будет забракован методом примитивной резолюции.

А в случае примера, приведенного на рис. 38, легко видеть, что все применения правила резолюции являются примитивными. Они подчиняются обоим эти ограничениям – ограничениям на присваивание и ограничением на ветвление.

Вывод.

Если мы ограничим себя классом линейно-ветвящихся программ, то никаких проблем в дедуктивном синтезе нет. Хорошо известный и многократно реализованный метод резолюций, который дополнен техникой предиката ANS и использованием примитивной резолюции, надежно и эффективно решает эту проблему.

6.3.5. Синтез циклических программ

Но проблема остается, и в ней скрыта причина принципиальной алгоритмической неразрешимости массовой проблемы автоматического синтеза программ – проблема синтеза циклов.

На самом деле не стоит сдаваться раньше времени. Алгоритмическая неразрешимость массовой проблемы – это, конечно, категорический запрет, но в самой его формулировке есть намек на обходной путь. Намек кроется в слове «массовая». Стоит изменить массовость, сузить рассматриваемый класс, и алгоритмическая неразрешимость может испариться без следа.

Пример.

Сейчас мы синтезируем циклическую программу – функцию для вычисления факториала.

Для начала разберемся со спецификацией этой программы.

Что такое факториал? Это такая функция f, что $f(0) = 1$, и $f(x+1) = (x+1)f(x)$.

Потусловие R состоит в том, что y является факториалом от x, $R(x, y) := y = f(x)$. В этом случае получаются следующие исходные аксиомы (для наглядности справа выписаны исходные рекуррентные соотношения):

1. $R(0,1)$ $f(0)=1$
2. $\neg R(x,y) \vee R(x+1,(x+1)*y)$ $f(x+1)=(x+1)*f(x)$.

Чтобы синтезировать циклическую программу, нужно иметь какую-то аксиому, которая бы это умела делать. Эта аксиома совершенно необязательно должна находиться внутри исходной спецификации. Понятие цикла синтезатор может знать и так.

Например, следующая формула исчисления предикатов первого порядка описывает ограниченный вид математической индукции:

$$(\exists y\, R(0, y) \wedge \forall x\, (\exists y_1\, R(x, y_1) \rightarrow \exists\, y_2\, R(x+1, y_2))) \rightarrow \forall x\, \exists y\, R(x, y)$$

Но если некоторые аксиомы поступают из самой задачи, а некоторые заранее заготовлены и есть в синтезаторе, то почему бы не провести предобработку этих заранее заготовленных аксиом. Проведем предобработку аксиомы математической индукции и скажем, что в любом случае, когда она применяется, мы будем синтезировать программу строго определенной структуры (рис. 42)

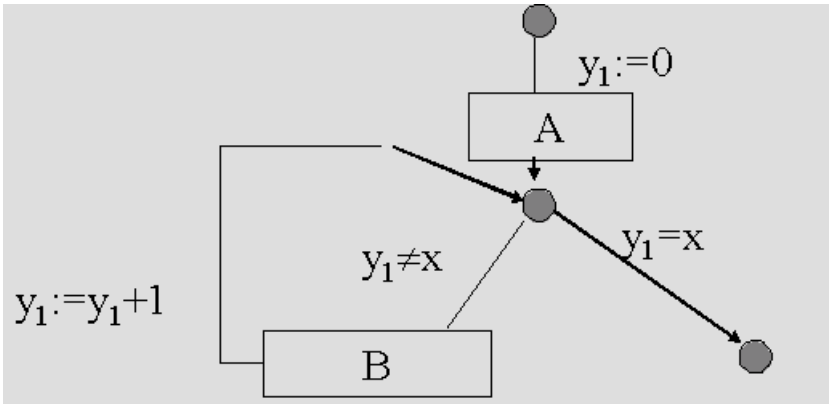


Рис. 42. Предопределенная структура циклической программы.

В ней будут уже заранее заготовлены некоторые проверки и некоторые присваивания, и будут два новых блока A и B, которые необходимо заполнить конкретными операторами при решении конкретной задачи, при синтезе конкретной циклической программы.

Это на самом деле известная блок-схема. Большая часть алгоритмов вычислительной математики имеют предопределенную структуру. Как правило итеративные методы вычислительной математики имеют указанную структуру.

Предобработка этой аксиомы такова. Для того, чтобы провести индуктивное доказательство, а из него извлечь итеративную программу, нужно доказать базу и доказать индуктивный переход принципа математической индукции. Это автоматически сгенерированные леммы.

3. $\neg R(0, y_2) \vee \text{ANS}(y_2)$
4. $R(y_1, y_2)$
5. $\neg R(y_1+1, y2) \vee \text{ANS(y2)}$

Предложение 4 – это наша цель, а 3 и 5 – это доказательство индуктивного перехода и доказательство базы соответственно. Появились две новые цели.

Теперь посмотрим, как будет проходить вывод по методу резолюций.

Синтез программы A будет производится за один шаг из предложений 1 и 3 подстановкой 1 вместо y_2:

6. ANS(1) из 1,3 т.е. A = y_2:=1

Синтез блока B производится за два шага:

7. $R(y_1+1,(y_1+1)*y_2) \vee \text{ANS}(y_2)$ из 2, 3
8. $\text{ANS}((y_1+1)*y_2)$ из 5, 7

Внутрь блока B попадает оператор, который должен попадать в тело цикла:

B = $y_2 := (y_1+1)*y_2$

В результате мы синтезировали одноэлементные дуги в нашем дереве. Заносим их в наше дерево и получаем итеративную программу вычисления факториала со всего лишь одним лишним оператором присваивания, который можно было бы и не использовать (рис. 40.):

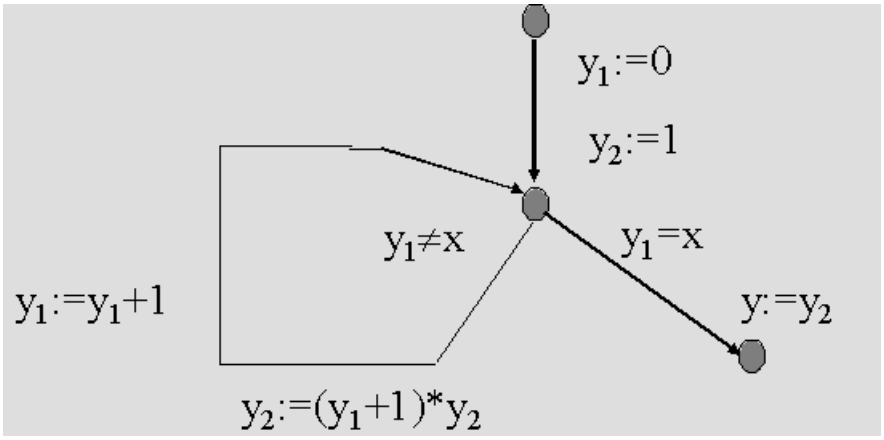


Рис. 43. Синтезированная программа вычисления факториала.

Рассмотренные примеры наглядно показывают, что на самом-то деле автоматически программировать достаточно легко. Но здесь есть небольшой секрет. Откуда мы догадались, что нужно применить именно эту форму индукции для синтеза факториала? Честный ответ на этот вопрос такой: мы это знали заранее и подогнали метод под нужный ответ. Главная причина алгоритмической неразрешимости синтеза программ в общем случае заключается в том, что невозможно автоматически догадаться, какую именно схему нужно применить в каждом конкретном случае. А схем этих бесконечно много.

Приведенный в примере метод действительно является методом автоматического синтеза циклических программ при условии, что синтезируемая программа содержится в классе программ, описываемых схемой на рис. 42.

Но если мы ограничиваем свой подход к синтезу программ определенным классом синтезируемых программ, то в этом случае все разрешимо. Можно построить системы, которые будут гарантировано правильно синтезировать программы из класса библиотек для С.

6.4. Структурный синтез программ

Рассмотрим еще один пример, относящий к пограничному случаю между дедуктивным и утилитарным методом синтезом программ. Этот пример очень поучителен. Идеи этого метода находят очень широкое применение даже в настоящее время.

6.4.1. Предпосылки

Исходные посылки таковы. Предыдущие примеры приводят нас к следующему, весьма пессимистическому наблюдению – труда на спецификацию и синтез программы затрачивается много, а синтезируем мы очень простые программы (факториал, вычисление знака числа и прочее). Конечно, этим заниматься не стоит.

В современном программировании из операторов никто программ не пишет, только студенты младших курсов на упражнениях по программированию. Нормальные программисты пользуются библиотеками, в которых есть мощные средства, которые многое способны сделать. Процедуры из этих библиотек содержат сотни, тысячи операторов, а программист ими пользуется как примитивными командами.

Нужно попытаться синтезировать программы не из операторов языка программирования, а из вызовов процедур имеющихся библиотек.
Почему мы считаем, что те процедуры, которые находятся в библиотеках правильные, и если их подставлять в синтезируемую программу, то эта программа будет работать правильно? Мы принимаем эту правильность на веру, верим производителям библиотек. Это утверждение, естественно, недоказуемо. В результате получается не гарантированно правильная программа, как в предыдущих примерах, а программа, правильная с точностью до правильности компонентов, из которых она построена.
Вообще говоря, все, что можно было запрограммировать в вычислительном смысле – запрограммировано. Остается только правильно воспользоваться тем, что уже готово. Мы не будем синтезировать вычисления, мы будем синтезировать только управляющую структуру.
Оказывается, что эта задача при наложении одного дополнительного ограничения решается очень красиво и эффективно. Само ограничение состоит в том, что в той предметной области, в которой мы проводим синтез программ, нет понятия времени. Если есть какие-то величины в предметной области, то они как-то связаны. И эта связь перманента. Например, при x равно 2, то y всегда равно 3. И эта связь не меняется с течением времени. У нас нет, и не будет операторов присваивания вида:

```
x := f(… , x, …)
```

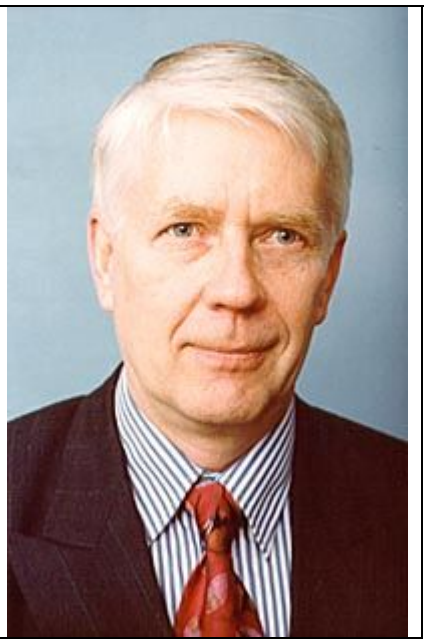
Присваивание никогда не будет пониматься как изменение значение переменной. Присваивание – это *определение* значения переменной. Значение переменной либо определено и не меняется никогда, либо не определено.
Это довольно сильное ограничение, но практика показывается, что оно выполняется в удивительно широком классе предметных областей.

Метод синтеза программ в предметных областях, удовлетворяющих указанному ограничению, называется структурный синтез программ.

6.4.2. История развития структурного синтеза

Структурный синтез программ был разработан Э.Х. Тыугу в середине 70-х годов. Позже стали использовать термин "семантические вычислительные сети" и "концептуальное программирование".

Тыугу Энн Харальдович, *род. 1935. – эстонский советский ученый. В составе Таллинской группы коллектива СТАРТ при Институте кибернетики АН ЭССР и СКБ ВТ разработал систему ПРИЗ – Программа Решения Инженерных Задач, в которой были реализованы его идеи структурного синтеза программ. Также группой были разработаны: рабочая станция ПИРС на процессоре Кронос; компилятор для Кроноса с языка С; операционная система Юникс для ПИРСа; среда программирования НУТ.*



Первоначальная идея родилась и первые пробные реализации были сделаны в начале 70-х в Институте кибернетики в Таллинне. В конце 70-х и до середины 80-х в Ленинграде под руководством С.С. Лаврова идея была значительно развита, как мне кажется, доведена до своего теоретического предела. На основе этой идеи в 80-е годы было сделано несколько десятков реализаций в разных организациях, как в форме систем автоматического синтеза программ общего назначения, так и в форме конкретных предметно-ориентированных пакетов прикладных программ, управляющие модули которых использовали данную идею. Теоретическую возможность синтеза программы за время, линейно зависящее от длины спецификации, обнаружил А. Диковский (Институт прикладной математики, Москва). В конце 80-х Ульман в своей фундаментальной книге по базам данных описал решение этой же задачи применительно к так называемым вычисляемым полям в базах данных (т.е. полям, значения которых не хранятся в базе, а динамически вычисляются на основе функциональных зависимостей от других полей). В настоящее время эта же деятельность продолжается под названием programming in constraints.

6.4.3. Семантическая вычислительная сеть.

Алгоритм будет сформулирован на основе представления данных, которое называется семантическая вычислительная сеть.

Семантическая вычислительная сеть – это двудольный ориентированный граф. Узлы одной доли называются модулями, узлы другой доли называются переменными.

Содержательно узлы первой доли – это функции нашей библиотеки, компоненты, которые умеют что-то делать. На рис. 44 они обозначены белым цветом. А серые узлы – это величины, некоторые из них заданы, некоторые мы должны определить.

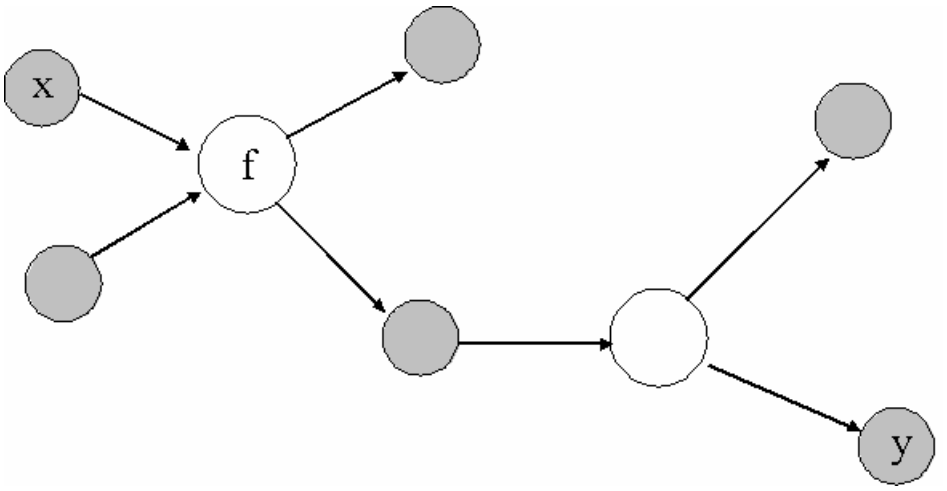


Рис. 44. Семантическая вычислительная сеть.

Другими словами, задано множество функций *F* и множество переменных предметной области *V*. Отношения между ними описаны множеством так называемых предложений вычислимости:

$$M = \{(X \to f \to Y)\}, \text{ где } f \in F, X \subset V, Y \subset V.$$

Каждое предложение вычислимости говорит, что если заданы значения таких-то переменных, то, применив данную функцию, можно получить значения других переменных.

Задача на семантической вычислительной сети формулируется следующим образом: задано множество *M*, задано множество переменных *U*, требуется определить множество переменных *W*.

6.4.4. Алгоритм прямой волны

Совершенно тривиальный алгоритм, типа алгоритма Уоршалла, который называется *алгоритм прямой волны*, достаточно легко решает эту задачу:

```
Proc Wave (U,W,M):P
     S := U
     P := ∅
L:   if W ⊂ S then return OK end if
     g := nil
     for f ∈  F do
          if X(f) ⊂ S & Y(f) ⊄ S then g:=f end if
     end for
     if g=nil then
          return fail
     else
          S := S ∪ Y(g)
          F := F\{g}
          P := P+g
          go to L
```

```
    end if
end proc
```

Суть алгоритма проста. Мы как бы пускаем волну, начиная с известных величин и определяя все, что можем определить, до тех пор, пока волна не накроет то, что нам действительно нужно определить.

Этот достаточно хороший алгоритм, и нетрудно сообразить, что работать он будет за $O(|M|^2)$ – это его трудоемкость.

Можно сделать и по-другому. Можно начать с W, и обратить стрелочки на графе. Применяя функцию в другую сторону, гнать волну не от известных к неизвестным, а наоборот. Очень полезно применить этот алгоритм дважды – туда и обратно. А потом взять пересечения в качестве ответа. Такое улучшение называется чисткой программы. После этого программа не будет содержать лишних, ненужных, неиспользуемых в окончательном результате вызовов компонентов.

6.4.5. Предварительная обработка МПО

Попробуем сделать то же самое не таким наивным прямолинейным способом, а с помощью небольшой предварительной обработки исходных данных. Задачи в результате станут решаться за линейное время.

Рассмотрим модель на основе следующей списочной структуры. Имеется основной двухсвязный список предложений вычислимости (рис. 45), в узлах этого списка выписаны функции f. У каждого функционального узла с каждой стороны двухсвязный список имен переменных (слева x – аргументы, справа y – результаты, которые получаются в результате применения каждой функции). Прошьем эту структуру еще группой списков. Для каждой переменной сделаем двухсвязный список ее вхождений в левые и в правые части. Список вхождений в левые части, то есть использование переменной в качестве аргумента функций будет называться X, а список вхождений в правые части, то есть использование переменной в качестве результата функций – Y.

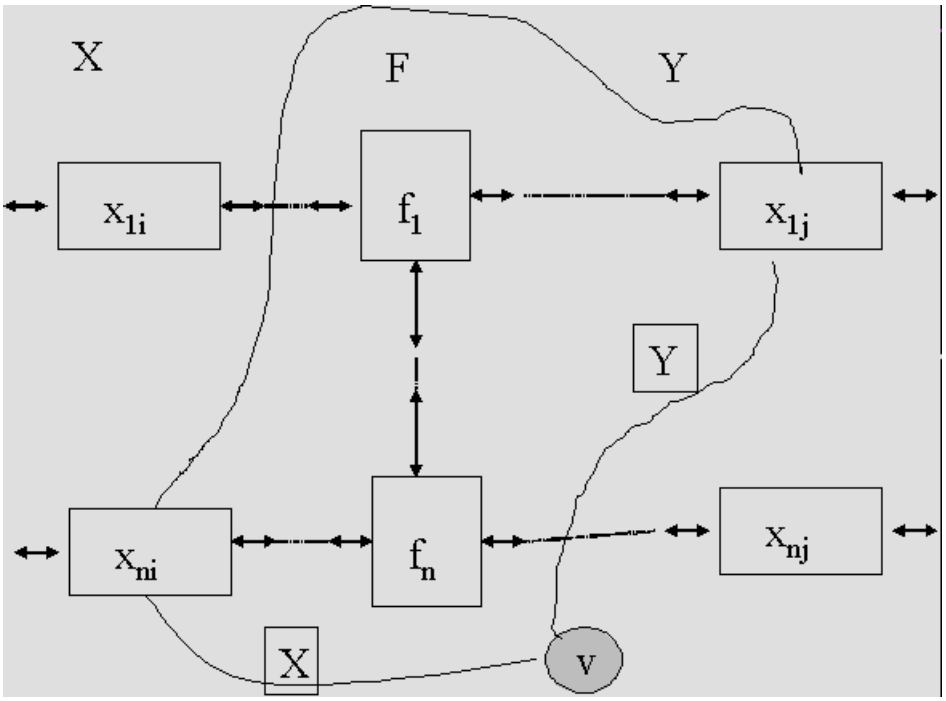


Рис. 45. Предобработка модели предметной области.

Для каждой переменной *v* в результате имеется возможность проследить все ее вхождения в эту сеть. Таким образом, каждая ячейка этой сети входит в два двусвязных списка и содержит четыре указателя.

Указанная структура данных и схема предобработки модели предметной области была предложена С.С. Лавровым.

6.4.6. Алгоритм Диковского-Ульмана

Алгоритм синтеза программ по предложениям вычислимости был независимо открыт несколькими авторами., Исторически первым его опубликовал Диковский, а потом независимо от Диковского через 10 лет опубликовал Ульман при решении задачи о функциональных зависимостях в базах данных. Ниже приведен сам алгоритм.

```
Proc DU(U, W, M)
     P:=nil
     G:=nil
     for u ∈ U do W:=W\{u}; S(u) end for
L:   if W=∅ then return P end if
     if G=nil then return fail end if
     g := car(G); G:=cdr(G); P:=P+g
     for y ∈ Y(g) do S(y); W:=W\{y} end for
     go to L
end proc
Proc S(u)
     for x ∈  X(u) do del(x)
```

```
            if X(f(x)) = nil then
                  G:=G+f          { функция применима }
            endif
      endfor
      for y ∈  Y(u) do del(y)
            if Y(f(y)) = nil then
                  G:=G\{f}        { функция бесполезна }
            endif
      endfor
end proc
```

Где `P` – синтезируемая программа; `G` – список применимых функций, `f(u)` – функциональный узел для узла `u`.

Основная процедура данного алгоритма – DU. Все переменные из числа известных нужно сначала выкинуть из числа неизвестных. Иногда это сильно помогает в решении, и задача оказывается тривиальной. После этого, пока у нас еще остались неизвестные, мы делаем следующее. Если неизвестных больше нет, то та программа, которая накопилась, уже является ответом. Если больше нечего применять, то происходит обрыв алгоритма – задача неразрешима. В противном случае мы берем первую функцию из списка применимых, остальные оставляем, добавляем эту функцию в синтезированную программу и выполняем процедуру `S` со всеми переменными, входящими в список y для выбранной нами функции.

Самое интересное в данном алгоритме – это процедура S. Она содержит два симметричных блока, отвечающих на вопросы:

1) что делать в той ситуации, когда мы узнали, что некоторая переменная может быть вычислена, и она входит куда-то в качестве аргумента;

2) что делать в той ситуации, когда мы знали, что некоторая переменная может быть вычислена, и она входит куда-то в качестве результата.

В первом случае мы смотрим, была ли эта переменная последней. Если да, то это означает, что она была последним неизвестным аргументом функции *f*, и функцию теперь можно применить. У нее все аргументы известны. Это примерно то же самое, что и управление по данным.

Во втором случае мы смотрим, была ли эта переменная последняя среди тех, которые данная функция вычисляет. Если да, то это означает, что такую функцию бесполезно применять, потому что она ничего нового вычислить не может.

И самое главное – почему этот алгоритм является линейный по длине модели. Он линейный, потому что, блуждая по этой списочной структуре (рис. 45), мы стираем все, что видим. А раз мы стираем все, что видим, значит это линейный алгоритм.

6.5. Индуктивный синтез программ

Если заданы протоколы выполнения некоторой программы, стоит попытаться индуктивным образом это обобщить и построить саму программу, которая, в частности, реализует и эти протоколы. Очень любопытный подход, но не очень популярный.

6.5.1. Многоточечные термы

Мы рассмотрим один пример, относящийся к классу индуктивного синтеза. Это так называемая теория многоточечных выражений. Целью этого примера являлся автоматический синтез программ, в том числе и циклических, но в довольно узком классе. Речь идет о целых числах, целочисленных массивах и вычислений с ними.

Задан некоторый алфавит A, содержащий числа – N и буквы – W. Числа на лексическом уровне отличаются от букв. Применяется также множество обычных разделителей U – разнообразные типы скобок ( ) [ ] { }. Особое значение имеют угловые скобки T – < > .

Символьная разность ÷ определяется следующим образом: $x \div y := 0$, если $x=y$; $x-y$, если $x, y \in N$ (т.е. если x и y –числа), а если x и y не числа – то символьная разность совершенно непонятно чем является, но при этом такая разность все равно может быть записана.

Вводится понятие многоточечный терм (…-терм): это выражение

$$D=\langle a_1\ a_2\ \ a_n \ldots b_1\ b_2\ \ b_n\rangle, \text{ где } a_i, b_i \in A \text{ и } \exists q \in Z\ \forall i\ 1 \le i \le n\ b_i = a_i \vee b_i \div a_i = q.$$

Здесь присутствует постоянная символическая разность q, называемая *фактором* многоточечного терма.

Значение многоточечного терма (…-терма)

$$V(D) := d_1^{\,0} \ldots d_n^{\,0}\ d_1^{\,1} \ldots d_n^{\,1} \ldots \ldots\ d_1^{\,|q|} \ldots d_n^{\,|q|}$$

где $d_i^{\,0} = a_i$, $d_i^{\,|q|} = b_i$ и $d_i^{\,j} \div d_i^{\,j-1} = 0$, если $b_i = a_i$; $d_i^{\,j} \div d_i^{\,j-1} = +1$, если $b_i \div a_i > 0$; $d_i^{\,j} \div d_i^{\,j-1} = -1$, если $b_i \div a_i > 0$.

Пример. Пусть D=<A(1)…A(4)>. Тогда V(D)=A(1)A(2)A(3)A(4).

В первых угловых скобках <A(1)…A(4)> – многоточечный терм, значение которого A(1)A(2)A(3)A(4). Фактор терма для этой записи q=1.

6.5.2. Многоточечные слова и выражения

Таким образом, имеются многоточечные термы. Давайте определим несколько надстроек над этим понятием – многоточечные слова и многоточечные выражения (обозначение …-слово и …-выражение).

Если X ∈ N или X ∈ W, то X …-слово.

Если X …-слово, то (X) {X} [X] …-слова.

Если X и Y …-слова, то XY …-слово.

Если X и Y …-слова, то <X…Y> …-слово, если выполнены условия многоточечного терма, т.е. постоянна символьная разность.

Для многоточечного выражения можно определить значение так же, как это было сделано для многоточечных термов; оно рекурсивно по термам.

Пример.

E = <<1…3>…<1…1>> <(4)(3)…(2)(1)>

V(E) = <1…3> <1…2> <1…1> (4)(3)(3)(2)(2)(1) = 1 2 3 1 2 1 (4)(3)(3)(2)(2)(1)

Введем кроме констант некоторый параметр k.

Слово с параметрами называется *многоточечным выражением*.

<<1…k>…<1…1>> <(k+1)(k)…(2)(1)>

6.5.3. Формальные примеры

Ниже приведен текст, который является протоколом выполнения программы. Он выглядит как многоточечное выражение:

<<if A(1)>A(2) then A(1)↔A(2);…
if A(k-1)>A(k) then A(k-1)↔A(k);> …
<if A(1)>A(2) then A(1)↔A(2);…
if A(1)>A(2) then A(1)↔A(2);>>

Формальный пример – значение многоточечного выражения для заданного значения *k*.

Значение предыдущего многоточечного выражения для *k*=3

if A(1)>A(2) then A(1)↔A(2);
if A(2)>A(3) then A(2)↔A(3);
if A(1)>A(2) then A(1)↔A(2);

Здесь формально выписан пример работы программы сортировки с использованием метода пузырька.

Таким образом, эти многоточечные выражения по существу не более, но и не менее чем другая форма записи циклическим программ с циклами по счетчикам.

6.5.4. Правила вывода

Введем два правила вывода: A_f и B. Подслово Y слова X называется (s,t) регулярным, если оно является значением многоточечного терма

$D = <x_1^1 x_s^1 \ldots x_1^t x_s^t>$

Пусть M(X) максимально входящее из N, m(X) – минимально входящее. То есть речь идет о самом большом и самом маленьком из чисел, входящих в многоточечное выражение.

f – некоторая целочисленная функция.

Правило A_f выглядит следующим образом: ищем самое левое максимальное (s,t)-регулярное подслово, такое, что t>f(M(X)) и заменяем его на многоточечный терм.

Правило B: числа x > (M(X)+m(X))/2 заменяем выражением k-c, где c = M(X)-x.

Правило B применимо, только если не применимо правило A_f.

Применение этих правил происходит следующим образом. Пусть имеется пример, содержащий повторяющиеся фрагменты. Находим самое левое, что можно обобщить в виде терма, на чем наблюдается постоянство символьной разности и обобщаем.

Если первое правило больше не удается применить, то вводим параметр и применяем второе правило.

6.5.5. Вывод многоточечного выражения из формального примера

Пусть задано слово:

X = A(1) A(2) A(3) A(4) A(1) A(2) A(3) A(1) A(2) A(1).

Зададим ограничение: *f* – целая часть квадратного корня, f(M(X))=2.

Вывод

A(1) A(2) A(3) A(4) A(1) A(2) A(3) A(1) A(2) A(1)
<A(1)…A(4)> A(1) A(2) A(3) A(1) A(2) A(1)
<A(1)…A(4)> <A(1)…A(3)> A(1) A(2) A(1)
<A(1)…A(4)> <A(1)…A(3)> <A(1)…A(2)> A(1)

<A(1)…A(4)> <A(1)…A(3)> <A(1)…A(2)><A(1)…A(1)>
<A(1)…A(k)><A(1)…A(k-1)><A(1)…A(2)><A(1)…A(1)>
< <A(1)…A(k)>… <A(1)…A(1)>>
A(1) A(2) – символьная разность q_1=1, A(2) A(3) – символьная разность q_1, A(3)A(4) – символьная разность q_1, A(4)-A(1) – символьная разность q_{-3}. Дальше алгоритм перестает действовать – прогрессия «сломалась». Но первую часть можно свернуть по правилу А. Продолжаем. Сворачиваем второй кусок.
Сначала работало правило А, пока не дошли до 5 строчки (см. выше). Дальше сворачивать уже нечего. Но теперь по правилу В можно ввести параметр. И сделать это два раза. В результате мы получаем очень хорошую циклическую запись программы, которая будет порождать такие протоколы.

6.5.6. Основной результат

Два многоточечных выражения называются *почти эквивалентными*, если их значения совпадают почти для всех значений *k*.

Т.е. мы будем строить не просто программу, а программу почти эквивалентную, которая почти всегда работает так, как нужно.
И основной результат можно сформулировать следующим образом:

$$\forall X\ \exists c\ \forall k>c\ (S_{Af,B}(V(X,k)\cong X)$$

Для любого многоточечного выражения существует такая константа, что для любого формального примера этого выражения, более длинного, чем заданная константа, результат индуктивного синтеза по данному примеру почти эквивалентен исходному многоточечному выражению.
Другими словами, для любой циклической программы существует такая константа, задающая число повторений цикла, что если записать протокол выполнения этой циклической программы длиннее, чем это число, то по этому протоколу можно построить другую циклическую программу, которая будет выдавать тот же ответ, что и исходная за исключением, возможно, конечного числа точек. Это и есть индуктивный синтез.

Предметный указатель

Литература

1. Н. Нильсен. Принципы искусственного интеллекта
2. Чень и Ли. Метод резолюций и автоматическое доказательство теорем
3. Лорьер Ж.-Л. Системы искусственного интеллекта. М.: Мир, 1991.
4. Лисков Б., Гатэг Дж. Использование абстракций и спецификаций при разработке программ. М.: Мир, 1989.
5. Приобретение знаний. М.: Мир, 1990.
6. Виноград Т. Программа, понимающая естественный язык. М.: Мир, 1976.
7. Искусственный интеллект: В 3-х книгах. М.: Радио и связь, 1990.
8. Ковальски Р. Логика в решении проблем. М.: Наука, 1990.
9. Ефимов Е.И. Решатели интеллектуальных задач. М.: Наука, 1982.