СИСТЕМЫ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА В АПК

приоритет2030
лидерами становятся

ТЕМА:
Модели представления знаний: алгоритмические, логические, сетевые и продукционные модели, сценарии

КОНСПЕКТ ЛЕКЦИИ

Преподаватель:

Аникуев Сергей Викторович
к.т.н., доцент, доцент кафедры электротехники, автоматики и метрологии.

приоритет2030^
лидерами становятся

Тема 1 Модели представления знаний: алгоритмические, логические, сетевые и продукционные модели, сценарии

Вопрос 1. Модели представления знаний

Вопрос 2. Решение задач

Вопрос 1. Модели представления знаний

Представление - это действие, делающее некоторое понятие воспринимаемым посредством рисунка, записи, языка или формализма. *Теория знаний* исследует связи между субъектом (изучающим) и объектом. *Знание* (в объективном смысле) - то, что известно (то, что мы знаем после изучения), т. е. - это совокупность сведений, образующих целостное описание, соответствующее некоторому уровню осведомленности об описываемом вопросе, предмете, проблеме и т.д.

Представление знаний - формализация знаний посредством фигур, записей или языков. Нас интересует формализации, воспринимаемые (распознаваемые) ЭВМ. Возникает вопрос о представлении знаний в памяти компьютера, т. е. о создании языков и формализмов представления знаний. Языки ИИ преобразуют наглядное представление (созданное посредством речи, изображением, естественным языком, вроде русского или английского, формальным языком, вроде алгебры или логики и т.д.) в пригодное для ввода и обработки на компьютере.

Представлению знаний присущ *пассивный аспект*: книга, таблица, заполненная информацией память. В ИИ подчеркивается *активный аспект* представления: *знать* должно стать активной операцией, позволяющей не только запоминать, но и извлекать воспринятые (приобретенные, усвоенные) знания (точнее, знания, которые запомнились) для рассуждения на их основе.

Наиболее часто знания подразделяются на декларативные и процедурные. *Процедурные знания* - это знания, хранящиеся в памяти интеллектуальной системы в виде описаний процедур, с помощью которых их можно получить. В виде процедурных знаний обычно описываются информация о предметной области, характеризующая способы решения задач в этой области, а также различные инструкции, методики, алгоритмы.

Знания декларативные - это знания, которые записаны в памяти интеллектуальной системы так, что они непосредственно доступны для использования после обращения к соответствующему полю памяти. В виде декларативных знаний обычно записывается информация о свойствах

предметной области, фактах, имеющих в ней место, и тому подобная информация. В отличие от процедурных знаний, отвечающих на вопрос "Как сделать X?", декларативные знания отвечают, скорее, на вопросы: "Что есть X?" или "Какие связи имеются между X и Y?", "Почему X?" и т.д.

На концептуальном уровне представления знаний наиболее распространены модели знаний в виде семантических сетей, фреймов и продукционных систем.

Продукция

Способ представления процедурных знаний в следующем наиболее общем виде:

(i); Q; P; C; AèB; N.

Здесь (i) — собственное имя (метка) продукции;

Q — сфера применения продукции, вычленяющая из предметной области некоторую ее часть, в которой знание, заключенное в продукции, имеет смысл;

P — предусловие, содержащее информацию об истинности данной продукция, ее приоритетности и т.п., используемую в стратегиях управления выводом для выбора данной продукции для исполнения;

C — условие, представляющее собой предикат, истинное значение которого разрешает применять на некотором шаге данную продукцию;

AèB — ядро продукции (интерпретация ядра может быть различной, например: "Если A истинно, то B истинно", "Если A имеется в базе знаний, то B надо внести в базу знаний", "Если A — текущая ситуация, то надо делать B" и т.п.);

N - постусловие продукции, содержащее информацию о том, какие изменения надо внести в данную продукцию или другие продукции, входящие в систему продукций, после выполнения данной продукции.

Примером реализации продукций является язык Пролог.

Семантическая сеть

Сеть (network) - это пятерка H=<A,B,P,P1,C>,

где A - множество вершин;

B - множество имен (весов) вершин;

P - множество дуг, соединяющих пары вершин;

P1 - множество отмеченных входных и выходных дуг;

С - множество весов (имен) дуг.

Семантическая сеть - это сеть, в вершинах которой информационные единицы, а дуги характеризуют отношения и связи между ними.

Для примера на рисунках приведены две семантические сети. Первая сеть (рис. 1.1) соответствует тексту: "В центре комнаты стоит стол. Слева от него окно. У стола глубокое удобное кресло. Недалеко от него столик с телефоном".

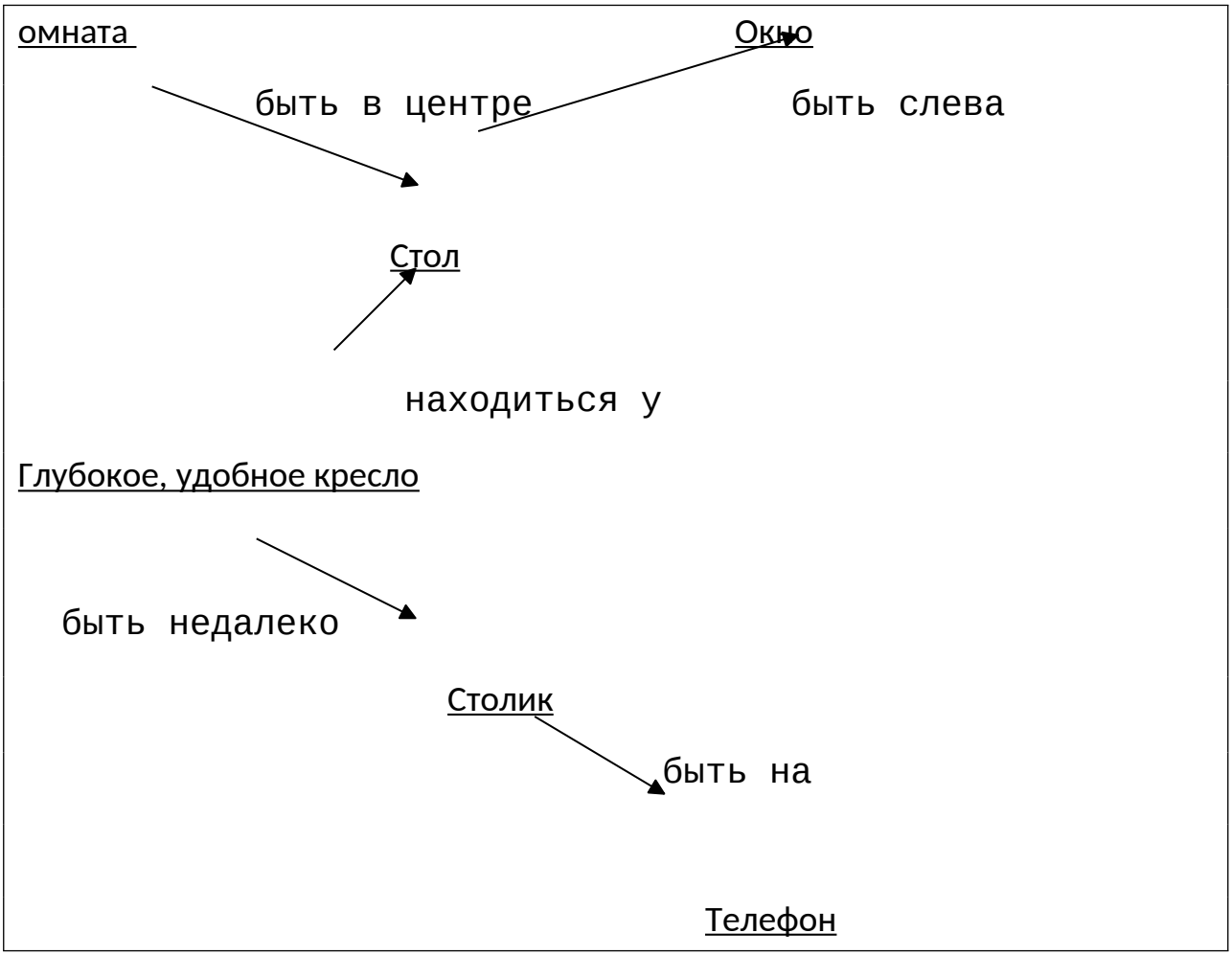


Рис. 1.1. Мебель в комнате

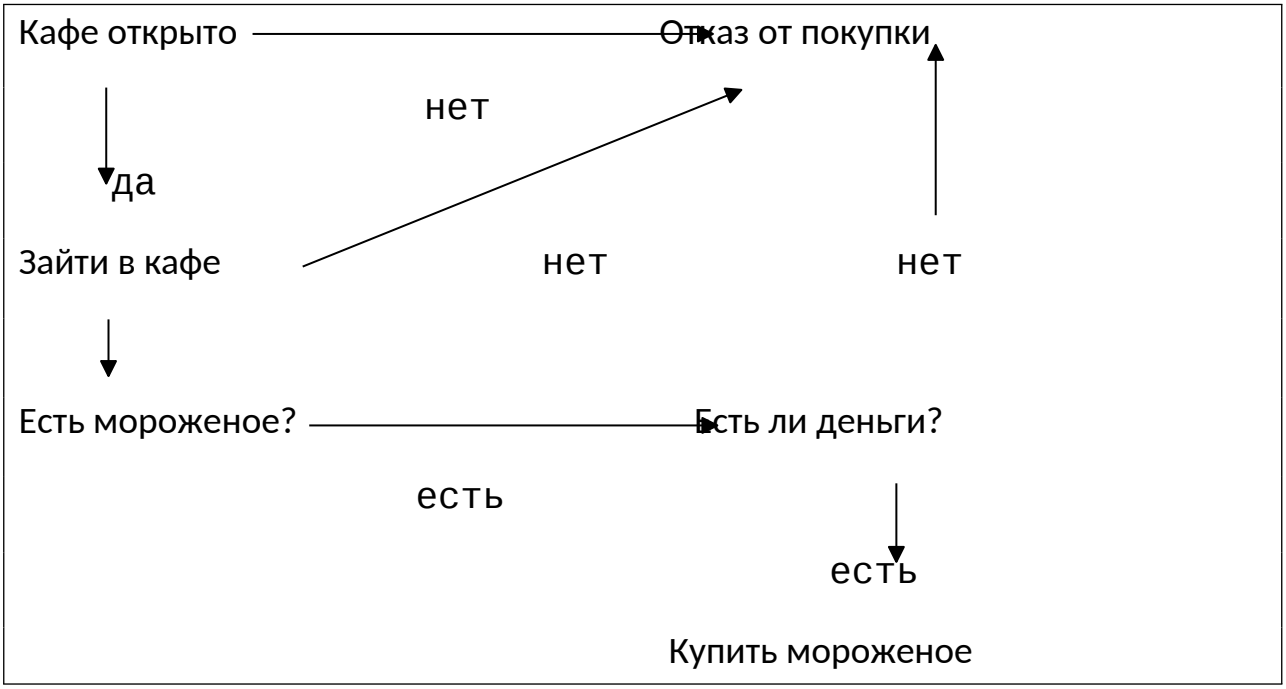


Рис. 1.2. Покупка мороженого в кафе

Вторая сеть (рис. 1.2) описывает набор процедур, необходимых для покупки мороженого в кафе. Если в вершинах первой сети находятся некоторые объекты (стол, комната и т.д.), то в вершинах второй сети названы процедуры. Дуги во второй сети не именованы, так как все они тут имеют одинаковый смысл: "перейти к процедуре".

Семантическая сеть является наиболее общей моделью представления знаний, но она осталась скорее теоретической моделью. Семантическую сеть можно реализовать без ограничений с помощью языка искусственного интеллекта Лисп.

Фреймы

Этот термин введен М. Минским для обозначения минимальной структуры, описывающей некоторое понятие или объект. Фрейм имеет имя (название) и состоит из частей, обычно называемых слотами; изображается фрейм в виде цепочки:

Фрейм = <слот 1><слот 2>...<слот N>.

Слот представляет собой пару атрибут (имя слота) — значение. В качестве значения могут выступать константные факты, выражения, содержащие переменные, ссылки на другие слоты, ссылки на фреймы и т. п.

Поэтому фрейм является рекурсивной структурой.

Рассмотрим в качестве примера фрейм "Битва". Цепочка этого фрейма выглядит так:

Битва = <кто?><с кем?><когда?><где?><результат>.

Представленный фрейм называется фреймом-прототипом. Во фреймах такого вида слоты имеют переменные значения. Например:

Битва = <Герой><Антигерой><утром><в чистом поле><победил>.

В этом случае, по крайней мере, слоты Герой и Антигерой - переменные, их значения могут уточняться. В результате такого уточнения получается фрейм-экземпляр:

Битва = <Царевич><Кощей Бессмертный><утром><в чистом поле><победил>.

Исключение из фрейма любого слота делает его принципиально неполным, иногда бессмысленным и не соответствующим названию фрейма.

Для автоматизированной обработки фреймов создан ряд языков, таких, например, как KPL, FPL. Близкое к понятию фрейма понятие объекта реализовано в объектно-ориентированном программировании.

Частным случаем фреймов являются *скрипты*, представляющие собой описание стереотипного сценария действия с участием определенных объектов. Скрипты связаны с текущей культурой и необходимым знанием для понимания таких предложений "Я вошел в ресторан, официантка принесла мне меню". Они могут вызывать другие скрипты и обладают большими, чем фреймы, возможностями для описания динамических аспектов знания.

Вывод знаний

Совокупность программных средств, обеспечивающих поиск, хранение, преобразование и запись в памяти компьютера сложно структурированных информационных единиц (знаний) называется *базой знаний*. В отличие от базы данных база знаний может содержать теоремы - частные случай продукционных правил с вполне определенными свойствами. *Вывод знаний* - получение новых информационных единиц из ранее известных.

Перечислим основные формы вывода знаний.

- Вывод абдуктивный
  Вывод на основании абдукции - правдоподобного заключения от частного к частному.

- Вывод вероятностный
  Вывод, при котором каждое выражение, используемое в нем, имеет оценку правдоподобия в виде вероятности того, что оно является истинным. При таком выводе применяются специальные процедуры для вычисления вероятности истинного значения результирующего выражения по вероятностям посылок, используемых при выводе.

- Вывод естественный
  Вывод, полученный на основании "здравого смысла". Эта форма вывода может либо соответствовать логическому выводу в некоторой формальной системе (но быть для человека очевидным), либо опираться

на соображения, которые не укладываются в строгие рамки формальной системы.

- Вывод индуктивный
  Вывод "от частного к общему". Позволяет на основании обобщения частных примеров некоторого явления выдвинуть гипотезу о существовании общей закономерности. В интеллектуальных системах, использующих индуктивный вывод, работают механизмы, позволяющие при формировании гипотезы приписывать ей оценку правдоподобия (например, вероятность того, что данная гипотеза является истинной). Индуктивный вывод является средством получения новых знаний в интеллектуальных системах.

- Вывод интуиционистский
  Вывод, характерный для интуиционистской логики, не использующий, в частности, закон снятия двойного отрицания и закон исключенного третьего.

- Вывод логический
  Последовательность рассуждений, приводящая от посылок к следствию с использованием аксиом и правил вывода. Такой вывод осуществляется в формальных аксиоматических системах, например, в логике предикатов первого порядка.

- Вывод на знаниях
  Вывод, использующий в качестве посылок выражения, хранящиеся в базе знаний. Вывод на знаниях может быть достоверным, если эти выражения являются достоверными, или правдоподобным, если они снабжены оценками правдоподобия. Как правило, процедуры вывода включают поиск необходимых фрагментов знаний для вывода, т.е. процедуру поиска по образцу.

- Вывод обратный
  Вывод, при котором поиск доказательства начинается с целевого утверждения. Выясняются условия, при которых целевое утверждение является выводимым. Эти условия принимаются за новые целевые утверждения, и процесс поиска продолжается. Вывод заканчивается,

когда все очередные условия оказываются аксиомами или процесс обрывается, не приведя к аксиомам. Обратный вывод реализован в алгоритме интерпретатора языка Пролог.

- Вывод по аналогии
  Вывод, основанный на перенесении рассуждения из одной области на другую область, похожую на исследованную. Если имеется вывод A Þ B и область, в которой определено A, гомоморфна области, где определена C, а область, где определено B, гомоморфна области, где определено D, то вывод A Þ B порождает вывод C Þ B. Вывод по аналогии есть частный случай правдоподобного вывода.

- Вывод правдоподобный
  Вывод, при котором каждый его шаг сопровождается вычислением оценки достоверности полученного утверждения. Частными случаями правдоподобного вывода являются, например, вывод вероятностный и вывод индуктивный.

- Вывод прямой
  Вывод, ведущий от исходных аксиом к целевому выражению. При прямом выводе из-за неоднозначности выбора применяемых аксиом и правил вывода образуется дерево решений и процесс нахождения цепочки, ведущей от исходных аксиом к целевому выражению, является переборным. Стандартной процедурой, используемой при обходе дерева решений, является процедура возврата — бектрекинг.

Изменение представлений

Поиск хорошего представления знаний в ходе решения задачи является почти всегда необходимым этапом на пути к решению.

Задача о четырех шахматных конях

Как за минимальное число ходов изменить на противоположное положение двух черных и двух белых коней? В задаче используется шахматная доска 3´3. Исходная позиция задачи о четырех конях

представлена на рис. 1.3. Кони ходят (перемещаются по доске) привычным образом. Центральная клетка имеет метку E.

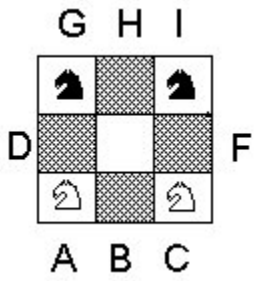


Рис. 1.3. Исходная позиция в задаче о 4 конях

При знакомстве с задачей нашим первым побуждением является сделать несколько попыток по перемещению коней на настоящей шахматной доске, но интуитивно мы понимаем, что, что имеется слишком много вариантов получения конечной позиции, причем при таком подходе плохо используется симметрия, присущая этой задаче. Пытаясь промоделировать условия задачи на уровне какого-то более общего представления, первое, о чем мы вспоминаем, это декартова система координат. Однако такое представление имеет существенный недостаток, связанный с трудностью записи перемещения фигур "ходом коня". Однако такие перемещения очень важны в нашем случае и именно они-то и должны быть представлены в удобной форме. Сама по себе шахматная доска почти не играет в задаче существенной роли, и ее физическое представление является только внешней поддержкой решения задачи, так как на самом деле единственно существенным является учет связей между ходами, сделанными при переходе от позиции к позиции. Мы знаем, что черный конь может переместиться на поле A за один ход с поля H или поля F. В свою очередь на поле H конь сможет попасть либо с того же поля A, либо с поля C, на поле C - с поля D, на него - с поля I, на поле I - с поля B, на него - с поля G и, наконец, на G - с поля F, начиная с которого конь сможет вновь попасть на

поле A. Этот маршрут в виде окружности, проходящей в указанном порядке через все позиции, изображен на рис. 1.4.

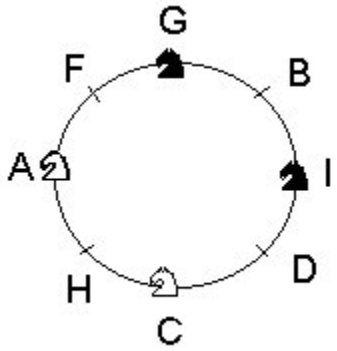


Рис. 1.4. Решение задачи о 4 конях

Представление, получаемое с помощью этой диаграммы, наглядно и удобно для использования. На рисунке не представлено поле E, но это соответствует реальной ситуации - ни при каком ходе коня ни в одной позиции оно недостижимо. Все другие ситуации отражены, и это позволяет использовать ту же кольцевую диаграмму и для других коней.

Новая формулировка задачи теперь состоит в том, что нужно так переместить по этой окружности черных коней на A и C, а белых коней на G и I, чтобы за один ход конь перемещался на одну позицию справа или слева по окружности. Решение задачи теперь практически очевидно: достаточно лишь повернуть на пол-оборота весь ансамбль фигур так, чтобы C попало на G, A - на I, G - на C и I - на A. Это вращение, которое соответствует четырем ходам каждого коня, или шестнадцати ходам всех фигур, дает решение задачи - минимальное число ходов для изменения исходной позиции на противоположную.

Изменения представления знаний во время решения задачи - основной путь ее решения.

В представлении знаний при решении математических задач существенным является то, о чем не говорится в учебниках по математике и на лекциях, а именно каким образом и как находят доказательство.

На самом деле способ действия математика включает три различные фазы:

1. *Понять теорему*, заданную на формальном языке, т.е. перевести ее в некоторое внутреннее представление.
2. *Доказать теорему* в этом внутреннем представлении, используя для этого все накопленные и адаптированные к этому внутреннему представлению знания.
3. *Перевести* найденное еще не полное доказательство в строгое математическое доказательство, вновь введя символическую формализацию обычного математического языка.

Мы будем использовать логический подход в изучении ИИ, хотя он и имеет большие ограничения (см. раздел 1.4. Психологическая теория интеллекта). Представлять знания мы будем с помощью формул логики предикатов первого порядка, и использовать логический вывод. Конкретным инструментом для программирования будет язык SWI-prolog (см. курс лекций по логическому программированию). Наш выбор логического подхода определяется достаточно быстрым получением реальных результатов для хорошо формализуемых задач.

Вопрос 2. Решение задач

Задача о кубиках. Общий метод решения задач

Задача состоит в выработке плана переупорядочивания кубиков, поставленных друг на друга, как показано на рис. 2.1. На каждом шагу разрешается переставлять только один кубик. Кубик можно взять только тогда, когда его верхняя поверхность свободна. Кубик можно поставить на стол, либо на другой кубик. Для того чтобы построить требуемый план, мы должны отыскать последовательность ходов, реализующую заданную трансформацию.

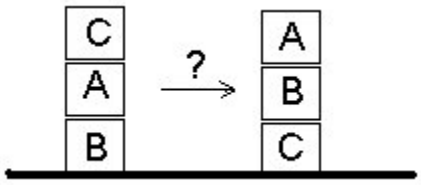


Рис. 2.1. Задача перестановки кубиков

Эту задачу можно представлять себе как задачу выбора среди множества возможных альтернатив. В исходной ситуации альтернатива всего одна: поставить кубик С на стол. После того, как кубик С поставлен на стол, мы имеем три альтернативы:

- поставить А на стол или

- поставить A на C, или
- поставить C на A.

Ясно, что альтернативу "поставить C на стол" не имело смысла рассматривать всерьез, так как этот ход никак не влияет на ситуацию.

Как показывает рассмотренный пример, с задачами такого рода связано два типа понятий:

- проблемные ситуации;
- разрешенные ходы или действия, преобразующие одни проблемные ситуации в другие.

Проблемные ситуации вместе с возможными ходами образуют направленный (ориентированный) граф, называемый *пространством состояний*. Пространство состояний для только что рассмотренного примера дано на рис. 2.2. Вершины графа соответствуют проблемным ситуациям, дуги - разрешенным переходам из одних состояний в другие. Проблема отыскания плана решения задачи эквивалентна проблеме построения пути между заданной начальной ситуацией ("стартовой" вершиной) и некоторой указанной заранее конечной ситуацией, называемой также *целевой вершиной*.

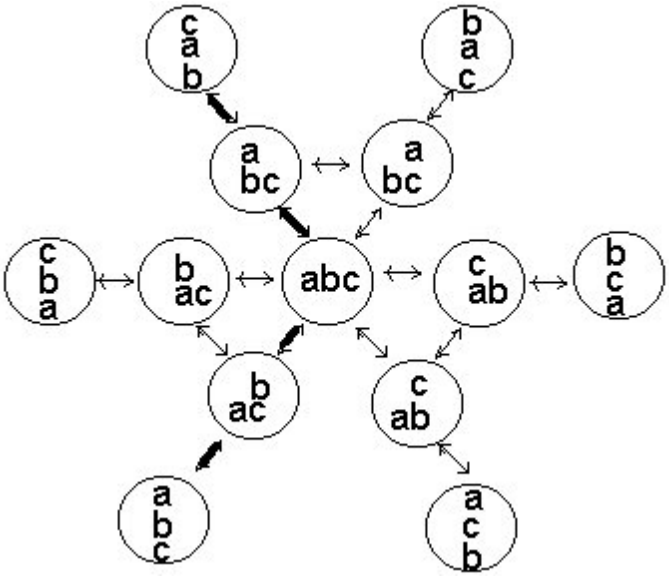


Рис. 2.2. Графическое представление задачи манипулирования кубиками. Выделенный путь является решением задачи

Нетрудно построить аналогичное представление в виде графа и для других популярных головоломок. Наиболее очевидные примеры - это задача о "ханойской башне" и задача о перевозке через реку волка, козы и капусты.

Пространство состояний некоторой задачи определяет "правила игры": вершины пространства состояний соответствуют ситуациям, а дуги - разрешенным ходам или действиям, или шагам решения задачи. Конкретная задача определяется

- пространством состояний,
- стартовой вершиной,
- целевым условием (т. е. условием, к достижению которого следует стремиться); "целевые вершины" - это вершины, удовлетворяющие этим условиям.

Каждому разрешенному ходу или действию можно приписать его стоимость. Например, в задаче манипуляции кубиками стоимости, приписанные тем или иным перемещениям кубиков, будут указывать нам на то, что некоторые кубики перемещать труднее, чем другие. В задаче о

коммивояжере ходы соответствуют переездам из города в город. Ясно, что в данном случае стоимость хода - это расстояние между двумя городами.

В тех случаях, когда каждый ход имеет стоимость, мы заинтересованы в отыскании решения минимальной стоимости. Стоимость решения - это сумма стоимостей дуг, из которых состоит "решающий путь" - путь из стартовой вершины в целевую. Даже если стоимости не заданы, все равно может возникнуть оптимизационная задача: нас может интересовать кратчайшее решение.

Прежде чем будут рассмотрены некоторые программы, реализующие классический алгоритм поиска в пространстве состояний, давайте сначала обсудим, как пространство состояний может быть представлено в прологовской программе.

Мы будем представлять пространство состояний при помощи отношения next(X,Y), которое истинно тогда, когда в пространстве состояний существует разрешенный ход из вершины X в вершину Y. Мы будем говорить, что Y - это *преемник* вершины X. Если с ходами связаны их стоимости, мы добавим третий аргумент, стоимость хода: next(X,Y,C). Эти отношения можно задавать в программе явным образом при помощи выбора соответствующих фактов. Однако такой принцип оказывается непрактичным и нереальным для тех типичных случаев, когда пространство состояний устроено достаточно сложно. Поэтому отношение следование next обычно определяется неявно, при помощи правил вычисления вершин-преемников некоторой заданной вершины. Другим вопросом, представляющим интерес с самой общей точки зрения, является вопрос о способе представления состояний, т.е. самих вершин. Это представление должно быть компактным, но в то же время оно должно обеспечивать эффективное выполнение необходимых операций, в частности операций вычисления вершин-преемников, а возможно и стоимостей соответствующих ходов.

Рассмотрим в качестве примера задачу манипулирования кубиками. Мы будем рассматривать общий случай, когда имеется произвольное число кубиков, из которых составлены столбики, - один или несколько. Число

столбиков мы ограничим некоторым максимальным числом, скажем 3, чтобы задача была интереснее. Такое ограничение, кроме того, является вполне реальным, поскольку рабочее пространство, которым располагает робот, манипулирующий кубиками, ограничено.

Проблемную ситуацию можно представить как список столбиков. Каждый столбик, в свою очередь, представляется списком кубиков, из которых он составлен. Кубики упорядочены в списке таким образом, что самый верхний список находится в голове списка. "Пустые" столбики изображаются как пустые списки. Таким образом, исходную ситуацию можно записать как терм [[c,a,b], [], []]. Целевая ситуация - это любая конфигурация кубиков, содержащая столбик, составленный из всех имеющихся кубиков в указанном порядке. Таких ситуаций три:

[[a,b,c], [], []]

[[], [a,b,c], []]

[[], [], [a,b,c]]

Отношение следование можно запрограммировать, исходя из следующего правила: ситуация S2 есть преемник ситуации S, если в S имеется два столбика C1 и C2, такие, что верхний кубик из C1 можно поставить сверху на C2 и получить тем самым S2. Поскольку все ситуации - это списки столбиков, правило транслируется на Пролог так:

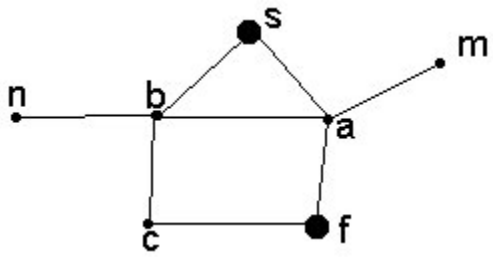
```
next(S, [C1,[Cub|C2]|T]):-          % переставить Cub на столбик C2
    select(S,[Cub|C1],S1),          %  найти первый столбик C1
    select(S1,C2,T).                % найти второй столбик C2
```

В нашем примере целевое условие имеет вид:

```
goal(S):- member([a,b,c],S).
```

Основные методы поиска

Существует много различных подходов к проблеме поиска решающего пути для задач сформулированных в терминах пространства состояний. В качестве примера графа, представляющего пространство состояний некоторой задачи, мы будем использовать граф на рис. 2.3. Поскольку мы считаем, что по любому ребру мы можем двигаться в обоих направлениях, то



стрелки на ребрах не указаны.

Рис. 2.3. Пространство состояний. s -начальная вершина, f - целевая вершина

При поиске пути из начальной в целевую вершину нам необходимо:

- использовать некоторую схему учета, позволяющую упорядоченным способом исследовать все возможные пути;
- не допускать циклов.

Для лучшего представления множества путей в графе полезно будет преобразовать граф в дерево (рис. 2.4), при этом корнем дерева является начальная вершина, а листья дерева - это целевые или тупиковые

вершины (тупики возникают в связи с нашим требованием не допускать циклов).

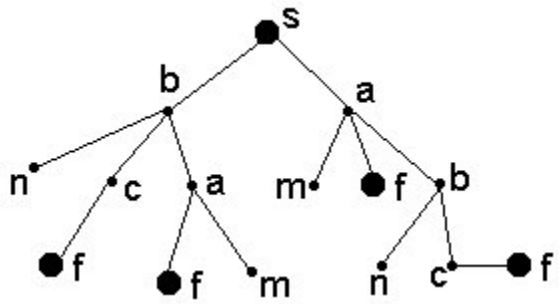


Рис. 2.4. Преобразование графа в дерево

Заметим, что число ярусов в полученном дереве не превосходит числа вершин в графе.

Поиск в глубину

Поиск в глубину основывается на следующей стратегии:

В каждой вершине выбирается какая-то определенная альтернатива, и ее изучение продолжается до тех пор, пока дальнейшее продвижение оказывается невозможным. Тогда процесс поиска возобновляется от ближайшей точки ветвления, имеющей неисследованные альтернативные варианты.

Поиск в глубину основывается на предположении "любой данный путь хорош, как и всякий другой". На рис. 2.5 показан порядок обхода вершин при поиске в глубину.

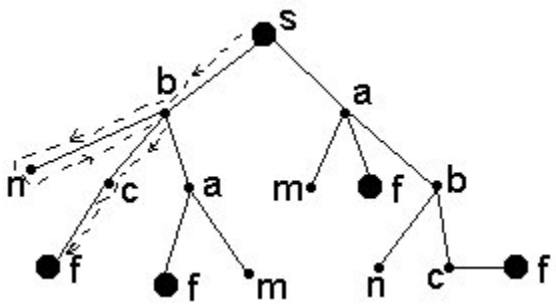


Рис. 2.5. Поиск в глубину

Поиск в глубину обладает следующими недостатками: а) во-первых, отыскивается не самый короткий путь; б) во-вторых, если в дереве есть бесконечные ветви и если такая бесконечная ветвь встретится - поиск никогда не закончится, даже если есть конечный путь в целевую вершину.

Поиск в глубину легко запрограммировать на Прологе.

```
% поиск в глубину

solve(Start,Solve):-          % Start - начальная вершина, Solve - искомый путь
        depth([],Start,Solve).

 depth(P,X,[X|P]):-
              goal(X).      % этот предикат проверяет, является ли вершина целевой

   depth(P,X,Solve):-
         next(X,X1),
         not(member(X1,P)),
         depth([X|P],X1,Solve).
```

Предикат depth использует первый аргумент для накопления пройденного пути. Вершины в пути перечисляются в обратном порядке.

Для графа на рис. 2.3 мы можем экономно определить отношение next:

```
'ребра'([[s,b],[s,a],[b,n],[b,c],[c,f],[a,m],[a,f],[a,b]]).
next(X,Y):-
      'ребра'(L),
```

(member([X,Y],L);member([Y,X],L)).

Добавим также к программе факт goal(f). Тогда имеем:

```
?- solve(s,X).
X = [f, c, b, s] ;
X = [f, a, b, s] ;
X = [f, a, s] ;
X = [f, c, b, a, s] ;
No
```

Вернемся к задаче о перестановке кубиков. Добавим предикат printList для удобной печати списка и предикат run для простоты запуска программы.

```
printList([]).
printList([H|T]):-
        write(H),nl,
        printList(T).

run:-
   solve([[c,a,b],[],[]],Solve),
   printList(Solve).

goal(S):- member([a,b,c],S).
```

Решим задачу:

```
?- run.
[[], [a, b, c], []]
[[a], [b, c], []]
[[b, a], [c], []]
[[], [c, b, a], []]
[[c], [b, a], []]
[[], [b, c], [a]]
[[b], [c], [a]]
[[], [c, b], [a]]
```

[[a], [c], [b]]
[[], [c, a], [b]]
[[c], [a], [b]]
[[], [a, c], [b]]
[[a], [b], [c]]
[[], [b, a], [c]]
[[b], [a], [c]]
[[], [c], [a, b]]
[[], [c], [a, b]]
[[c], [a, b], []]
[[a, c], [b], []]
[[], [b, a, c], []]
[[b], [a, c], []]
[[a, b], [c], []]
[[c, a, b], [], []]
Yes

У нас получилось поистине "ужасное" решение. Разберемся в чем причина. Во-первых, ситуации в найденном решении повторяются: например, состояния [[], [c], [b,a]], [[c], [b,a], []] и [[b,a], [c], []] в программе различаются. Это явилось следствием того, что в списке столбиков учитывается порядок. Для улучшения программы надо столбики рассматривать как элементы множества и заменить предикат member более сложным предикатом. Во-вторых, в решения встречаются два одинаковых состояния, идущих подряд

[[], [c], [a, b]]
[[], [c], [a, b]]

Это уже следствие недостаточно хорошего определения предиката next. Дело в том, что одним из состояний-преемников для [[], [c], [a, b]] является состояние [[c], [], [a, b]], которое предикат next выдает в виде [[], [c], [a, b]]. Здесь снова при программировании next надо учесть, что порядок перечисления столбиков в состоянии для нас не важен.

Если известна верхняя граница длины решающего пути, то можно ограничить глубину поиска.

```
% Поиск в глубину с ограничением глубины
solve(Start,Solve):-          % Start - начальная вершина, Solve - искомый путь
    depth([],Start,Solve).
```

```
depth(P,X,[X|P]):-
        goal(X),
        length([X|P],N),nl,
        write('Нашли решение за '),write(N),write(' шагов '),nl.
depth(P,X,Solve):-
        maxlength(Max),
        length(P,N),
        N+1<Max,
        next(X,X1),
        not(member(X1,P)),
        depth([X|P],X1,Solve).

% maxlength(N) -> N - максимальная глубина;
maxlength(10).
```

Теперь, чтобы решить задачу (при поиске в глубину с ограничением 10)

достаточно запустить цель

```
?- run.
```

Нашли решение за 10 шагов

[[], [a, b, c], []]

[[], [b, c], [a]]

[[b], [a], [c]]

[[], [c], [a, b]]

[[c], [a, b], []]

[[a, c], [b], []]

[[], [b, a, c], []]

[[b], [a, c], []]

[[a, b], [c], []]

[[c, a, b], [], []]

Yes

Поиск в глубину наиболее адекватен рекурсивному стилю программирования, принятому в Прологе. Причина этого состоит в том, что, обрабатывая цели, пролог-система сама просматривает альтернативы именно в глубину.

Поиск в ширину

При поиске в ширину целевая вершина сначала отыскивается среди всех вершин, расположенных на данном уровне, прежде чем будут исследованы ветви, отходящие вниз от этих вершин (рис. 2.6). Иными словами, сначала ищем решение среди путей длины один, потом - длины два и т.д.

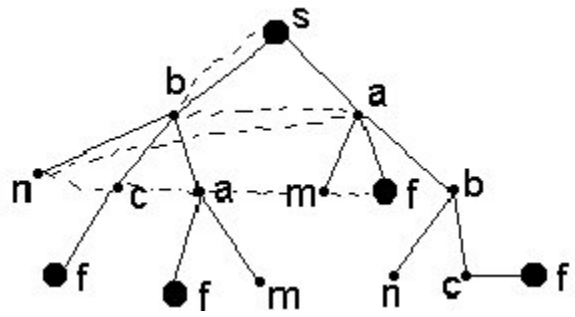


Рис. 2.6. Поиск в ширину

Очевидно, что этот поиск применим, даже если дерево бесконечно или практически бесконечно. К недостаткам этого поиска можно отнести неэффективность: если b - среднее число альтернатив для каждой внутренней вершины, то число путей длины n равно в среднем b^n.

Поиск в ширину программируется на Прологе немного сложнее.

```
% поиск в ширину
solve(Start,Solve):-
       width([[Start]],Solve).

width([[X|P]|_],[X|P]):- goal(X).
width(Ps,Solve):-
          gener(Ps,Npath),
          width(Npath,Solve).

newPath([],_,[]).
newPath([X|T],L,[[X|L]|LL]):-
           newPath(T,L,LL).

postList(X,L,K):-
     findall(Y, (next(X,Y),not member(Y,L)),K).

gener([[X|L]|T],Npath):-
             postList(X,L,K),
             newPath(K,[X|L],ZZ),
             append(T,ZZ,Npath).
```

Предикат width(LL,S) использует аргумент LL для хранения всех начатых и еще не рассмотренных путей. LL представляет из себя список путей - список списков. Рассматриваемый в текущий момент путь является головой списка LL. Если этот путь приходит в целевую вершину, то решение найдено (это первое правило для width). Иначе применяется второе правило для width: создается новый список путей - кандидатов к рассмотрению - и рекурсивно вызывается снова width.

Предикат gener([[X|L]|T],Npath) удлиняет на одну вершину первый путь [X|L] из списка путей-кандидатов и множество удлиненных путей добавляется в конец списка путей-кандидатов. Используемый в нем вызов предиката postList(X,L,K) создает список K вершин-преемников вершины X, не принадлежащих списку L.

Проверим, как работает программа. Сначала поиск в ширину в графе на рис. 2.3:

?- solve(s,Solve).

Solve = [f, a, s] ;

Solve = [f, c, b, s] ;

Solve = [f, a, b, s] ;

Solve = [f, c, b, a, s] ;

No

Для задачи о кубиках поиск в ширину сразу выдает самое короткое

решение:

?- run.

[[], [a, b, c], []]

[[], [b, c], [a]]

[[b], [a], [c]]

[[a, b], [c], []]

[[c, a, b], [], []]

Yes

Поиск в глубину и поиск в ширину относятся к стратегиям полного перебора. Эффективность поиска повышается, если упорядочить ветви, идущие от каждой вершины, - в первую очередь при переборе выбирать наиболее перспективные ветви. Мы используем это, скажем, при подъеме в гору, выбираем тропинки, идущие вверх. Другой пример: если вам надо на автомобиле пересечь незнакомый город в направлении с севера на юг, то вы предпочитаете ехать по широким улицам, идущим близко к этому направлению. Такая стратегия называется *стратегией наискорейшего подъема (спуска)*.

Если мы отказываемся от полного перебора и отбрасываем некоторые, на наш взгляд, неперспективные ветви, то такой поиск называется *эвристическим*. Эвристический поиск быстрее находит решение, хотя и может быть неудачным. Пример: если мы должны выйти из леса в город, то следует предпочесть асфальтированные дороги проселочным.

Сведение задач к подзадачам. И/ИЛИ-графы

Представление задач в виде И/ИЛИ-графов

Мы рассмотрели подход к решению задач, основанный на поиске пути в графе пространства состояний. Однако для некоторых категорий задач представление в форме И/ИЛИ-графа является более естественным. Такое представление основано на разбиении задач на подзадачи. Разбиение на подзадачи дает преимущества в том случае, когда подзадачи взаимно независимы, а, следовательно, и решать их можно независимо друг от друга.

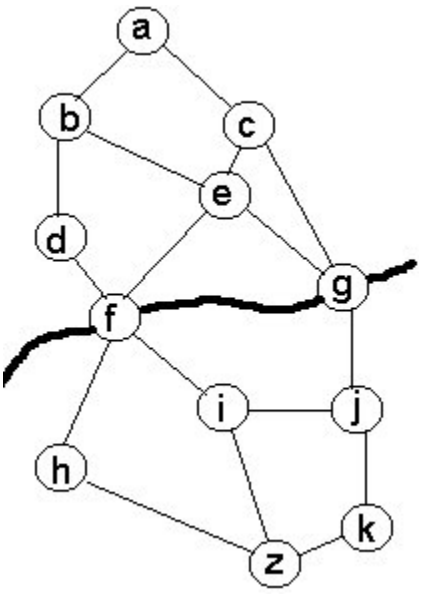


Рис. 2.7. Поиск маршрута из a в z на карте дорог.

Через реку можно переправиться в городах f и g.

И/ИЛИ-представление этой задачи показано на рис. 2.8.

Проиллюстрируем это на примере. Рассмотрим задачу отыскания на карте дорог маршрута между двумя заданными городами, как показано на рис. 2.7. Не будем учитывать длину путей. Разумеется, эту задачу можно сформулировать как поиск пути в пространстве состояний. Соответствующее пространство состояний выглядело бы в точности, как карта рис. 2.7: вершины соответствуют городам, дуги - непосредственным связям между городами. Тем не менее, давайте построим другое представление, основанное на естественном разбиении этой задачи на подзадачи. На карте рис. 2.7 мы видим также реку. Допустим, что переправиться через нее можно только по двум мостам: один расположен в городе f, другой - в городе g. Очевидно, что искомый маршрут обязательно должен проходить через один из мостов, а значит, он должен пройти либо через f, либо через g. Таким образом, мы имеем две главных альтернативы:

Для того чтобы найти путь из a в z, необходимо найти одно из двух:

(1) путь из a в z, проходящий через f, или

(2) путь из a в z, проходящий через g.

Теперь каждую из этих двух альтернативных задач можно, в свою очередь, разбить следующим образом:

(1) Для того, чтобы найти путь из a в z через f, необходимо:

1.1 найти путь из a в f и

1.2 найти путь из f в z.

(2) Для того, чтобы найти путь из a в z через g, необходимо:

2.1 найти путь из a в g и

2.2. найти путь из g в z.

Итак, мы имеем две главные альтернативы для решения исходной задачи: (1) путь через f или (2) путь через g. Далее, каждую из этих альтернатив можно разбить на подзадачи (1.1 и 1.2 или 2.1 и 2.2 соответственно). Здесь важно то обстоятельство, что каждую из подзадач в обеих альтернативах можно решать независимо от другой. Полученное разбиение исходной задачи можно изобразить в форме *И/ИЛИ -графа* (рис. 2.8).

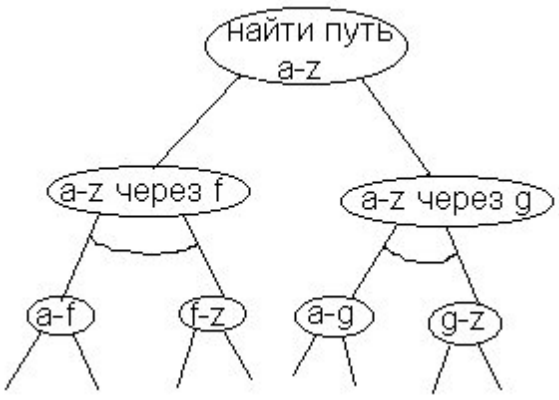


Рис. 2.8. И/ИЛИ-представление задачи поиска маршрута рис.2.7

Вершины соответствуют задачам или подзадачам, полукруглые дуги означают, что все (точнее, обе) подзадачи должны быть решены.

Обратите внимание на полукруглые дуги, которые указывают на отношение И между соответствующими подзадачами. Граф, показанный на рис. 2.8 - это всего лишь верхняя часть всего И/ИЛИ-дерева. Дальнейшее разбиение подзадач можно было бы строить на основе введения дополнительных промежуточных городов.

Какие вершины И/ИЛИ-графа являются целевыми? Целевые вершины - это тривиальные, или "примитивные" задачи. В нашем примере такой подзадачей можно было бы считать подзадачу "найти путь из а в с", поскольку между городами а и с на карте имеется непосредственная связь.

Рассматривая наш пример, мы ввели ряд важных понятий. И/ИЛИ-граф - это направленный граф, вершины которого соответствуют задачам, а дуги отношениям между задачами. Между дугами также существуют свои отношения. Это отношения И и ИЛИ, в зависимости от того, должны ли мы решить только одну из задач-преемников или же несколько из них (рис. 2.9). В принципе из вершины могут выходить дуги, находящиеся в отношении И вместе с дугами, находящимися в отношении ИЛИ. Тем не менее, мы будем предполагать, что каждая вершина имеет либо только И-преемников, либо только ИЛИ-преемников; дело в том, что в такую форму

можно преобразовать любой И/ИЛИ-граф, вводя в него при необходимости вспомогательные ИЛИ-вершины. Вершину, из которой выходят только И-дуги, называют И-вершиной; вершину, из которой выходят только ИЛИ-дуги, - ИЛИ-вершиной.

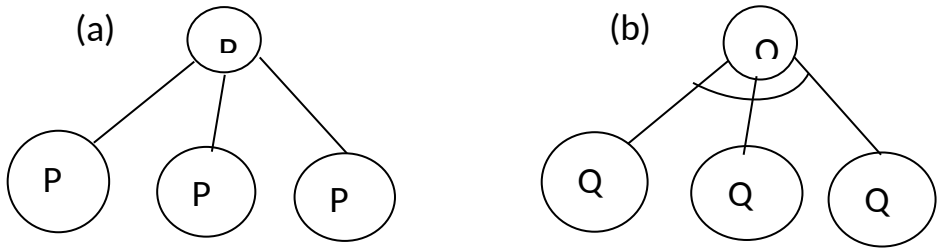


Рис. 2.9. (a) Решить P - это значит решить P1или P2 или ...
(b) Решить Q - это значит решить Q1 и Q2 и ... .

Когда задача представлялась в форме пространства состояний, ее решением был путь в этом пространстве. Что является решением в случае И/ИЛИ-представления? Решение, конечно, должно включать в себя все подзадачи И-вершины. Следовательно, это уже не путь, а дерево. Такое решающее дерево T определяется следующим образом:

- исходная задача P - корень дерева T;
- если P является ИЛИ-вершиной, то в T содержится только один из ее преемников (из И/ИЛИ-графа) вместе со своим собственным решающим деревом;

- если P - это И-вершина, то все ее преемники (из И/ИЛИ-графа) вместе со своими решающими деревьями содержатся в T.

Поиск в И/ИЛИ-графах

Простейший способ организовать поиск в И/ИЛИ-графах средствами Пролога - это использовать переборный механизм, заложенной в самой пролог-системе. Оказывается, что это очень просто сделать, потому что процедурная семантика Пролога это и есть не что иное, как поиск в И/ИЛИ-графе. Например, И/ИЛИ-граф задачи на рис. 2.7 можно описать при помощи следующих предложений (предикат a-z соответствует задаче "найти путь из a в z", предикат a-z/f соответствует задаче "найти путь из a в z через f" и т.д.):

```
a-z:-a-z/f.       % задача a-z - ИЛИ-вершина с двумя преемниками
a-z:-a-z/g.       % a-z через f и a-z через g

a-z/f:- a-f,f-z.  % задача a-z/f - И-вершина с двумя преемниками a-f и f-z

a-z/g:-a-g,g-z.
a-f:-a-f/b.
a-f:-a-f/c.
a-f/b:-a-b,b-f.
a-f/c:-a-c,c-f.
b-f:-b-d,d-f.
c-f:-c-e,e-f.
f-z:-f-z/h.
f-z:-f-z/i.
f-z/h:-f-h,h-z.
f-z/i:-f-i,i-z.
/* пропущены правила
```

```
   для a-g и g-z
*/
a-b. b-d. d-f. a-c. c-e. e-f. f-h. h-z. f-i. i-z. % "тривиальные" задачи
```

Для того чтобы узнать, имеет ли эта задача решение, нужно просто спросить:

```
?- a-z.
```

Получив этот вопрос, пролог-система произведет поиск в глубину в И/ИЛИ-дереве и, после того как найдет решающее дерево, ответит Yes.

За простоту такого метода программирования приходится расплачиваться: мы не получаем явно решающего дерева. Но этот недостаток исправим - надо определить собственную процедуру поиска в глубину для И/ИЛИ-дерева.

Игры и минимаксный принцип

Формулировка игровых задач в терминах И/ИЛИ-графов

Такие игры, как шахматы или шашки, естественно рассматривать как задачи, представленные И/ИЛИ-графами. Игры такого рода называются играми двух лиц с полной информацией. Будем считать, что существует только два возможных исхода игры: ВЫИГРЫШ и ПРОИГРЫШ. (Об играх с тремя возможными исходами - ВЫИГРЫШ, ПРОИГРЫШ и НИЧЬЯ, можно также говорить, что они имеют только два исхода: ВЫИГРЫШ и НЕВЫИГРЫШ). Так как участники игры ходят по очереди, мы имеем два

вида позиций, в зависимости от того, чей ход. Давайте условимся называть участников игры "игрок" и "противник", тогда мы будем иметь следующие два вида позиций: позиция с ходом игрока ("позиция игрока") и позиция с ходом противника ("позиция противника"). Допустим также, что начальная позиция P - это позиция игрока. Каждый вариант хода игрока в этой позиции приводит к одной из позиций противника Q1, Q2, Q3, ... (рис. 2.10).

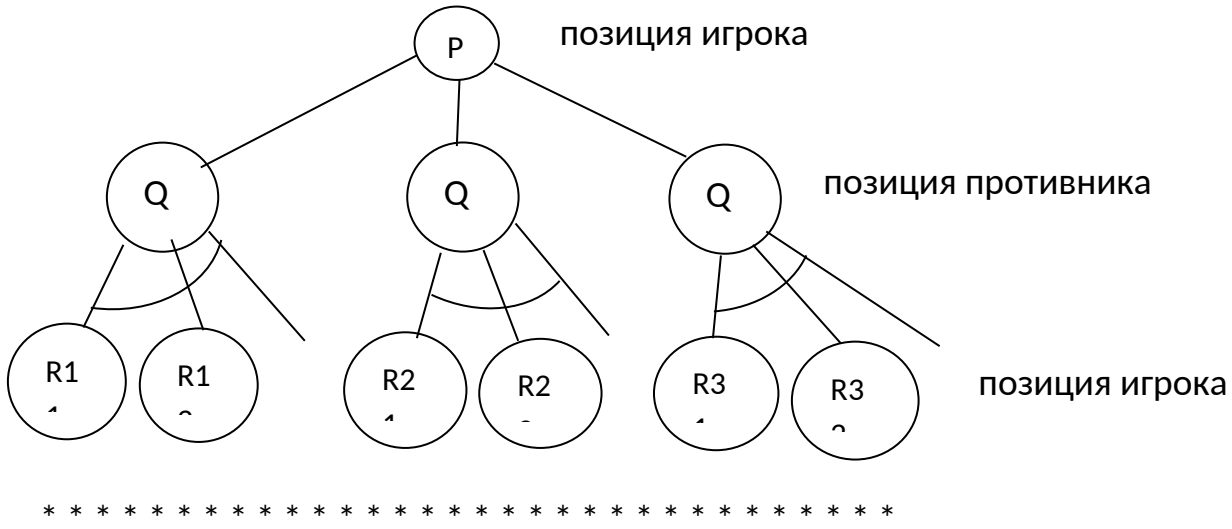


Рис. 2.10. Формулировка игровой задачи для игры двух лиц в форме И/ИЛИ-дерева; участники игры: "игрок" и "противник"

Далее, каждый вариант хода противника в позиции Q1 приводит к одной из позиций игрока R11, R12, ... В И/ИЛИ-дереве, показанном на

рис. 2.10, вершины соответствуют позициям, а дуги - возможным ходам. Уровни позиций игрока чередуются в дереве с уровнями позиций противника. Для того чтобы выиграть в позиции P, нужно найти ход, переводящий позицию P в выигранную позицию Qi (при некотором i). Таким образом, игрок выигрывает в позиции P, если он выигрывает в Q1 или Q2, или Q3, или ... . Следовательно, P - это ИЛИ-вершина. Для любого i, позиция Qi - это позиция противника, поэтому если в этой позиции выигрывает игрок, то он выигрывает и после каждого варианта хода противника. Другими словами, игрок выигрывает в Qi, если он выигрывает во всех позициях Ri1 и Ri2 и ... . Таким образом, все позиции противника - это И-вершины. Целевые вершины - это терминальные (окончательные) позиции, выигранные согласно правилам игры, например, позиции, в которых король противника получает мат. Позициям проигранным соответствуют задачи, не имеющие решения. Для того чтобы решить игровую задачу, мы должны построить решающее дерево, гарантирующее победу игрока независимо от ответов противника. Такое дерево задает полную стратегию достижения выигрыша: для каждого возможного продолжения, выбранного противником, в дереве стратегии есть ответный ход, приводящий к победе.

Ниже приводится простая программа, которая определяет, является ли некоторая позиция игрока выигранной.

```
'выигранная'(P):-
        'терм_выигранная'(P). % терминальная выигранная позиция

'выигранная'(P):-
        not 'терм_проигранная'(P), % не терминальная проигранная
                                   % позиция
```

```
        'ход'(P,P1),        % разрешенный ход из позиции P в позицию P1
        % ни один из ходов противника не ведет к не-выигрышу
        not('ход'(P1,P2),
            not 'выигранная'(P2)).
```

Здесь правила игры встроены в предикат 'ход'(P,P1), который порождает все разрешенные ходы, а также в предикаты 'терм_выигранная'(P) и 'терм_проигранная'(P), которые распознают терминальные позиции, являющиеся, согласно правилам игры, выигранными или проигранными. В последнем из правил программы, содержащем двойное отрицание, говорится: не существует хода противника, ведущего к не выигранной позиции. Другими словами, все ходы противника приводят к позициям, выигранным с точки зрения игрока.

Программа, которую мы составили, демонстрирует основные принципы программирования игр. Но практически приемлемая реализация таких сложных игр, как шахматы или го, потребовала бы привлечения значительно более мощных методов. Огромная комбинаторная сложность этих игр делает наш наивный переборный алгоритм, просматривающий дерево вплоть до терминальных игровых позиций, абсолютно непригодным. Для шахмат, например, пространство поиска имеет астрономические размеры - около 10^{120} позиций.

Минимаксный принцип

Поскольку полный просмотр игрового дерева в большинстве игр невозможен были разработаны другие методы, предусматривающие просмотр только части дерева игры. Среди этих методов существует стандартный метод поиска, используемый в игровых (особенно в шахматных) программах и основанный на *минимаксном принципе*. Дерево игры (И/ИЛИ-граф) просматривается только вплоть до некоторой глубины (обычно на несколько ходов), а затем для всех концевых вершин дерева поиска вычисляются оценки позиций при помощи некоторой оценочной

функции. Идея состоит в том, чтобы, получив оценки этих терминальных поисковых вершин, не продвигаться дальше и получить тем самым экономию времени. Далее оценки терминальных позиций распространяются вверх по дереву поиска в соответствии с минимаксным принципом. В результате все вершины дерева поиска получают свои оценки. И, наконец, игровая программа, участвующая в некоторой реальной игре, делает свой ход - ход, ведущий из исходной (корневой) позиции в наиболее перспективного (с точки зрения оценки) ее преемника.

Обратите внимание на то, что мы здесь делаем определенное различие между "деревом игры" и "деревом поиска". Дерево поиска - это только часть дерева игры (его верхняя часть), т. е. та его часть, которая была явным способом порождена в процессе поиска. Таким образом, терминальные поисковые позиции совсем не обязательно должны совпадать с терминальными позициями самой игры.

Очень много зависит от оценочной функции, которая для большинства игр, представляющих интерес, является приближенной эвристической оценкой шансов на выигрыш одного из участников игры. Чем выше оценка, тем больше у него шансов выиграть и чем ниже оценка, тем больше шансов на выигрыш у его противника. Поскольку один из участников игры всегда стремиться к высоким оценкам, а другой - к низким, мы дадим им имена МАКС и МИН соответственно. МАКС всегда выбирает ход с максимальной оценкой; в противоположность ему МИН всегда выбирает ход с минимальной оценкой. Пользуясь этим принципом (минимаксным принципом) и зная значения оценок для всех вершин "подножья" дерева поиска, можно определить оценки всех остальных вершин дерева. На рис. 2.11 показано, как это делается. На этом рисунке видно, что уровни позиций с ходом МАКСа чередуются с уровнями позиций с ходом МИНа. Оценки вершин нижнего уровня определяются при помощи оценочной функции. Оценки всех внутренних вершин можно определить, двигаясь снизу вверх от уровня к уровню, пока мы не достигнем корневой вершины. в результате, как видно на рис. 2.11, оценка корня оказывается равной 4, и, соответственно, лучшим ходом МАКСа из позиции a - a-b. Лучший ответ МИНа на этот ход - b-d, и т.д. Эту последовательность ходов называют также *основным*

вариантом. Основной вариант показывает, какова "минимаксно-оптимальная" игра для обоих участников. Обратите внимание на то, что оценки всех позиций, входящих в основной вариант совпадают.

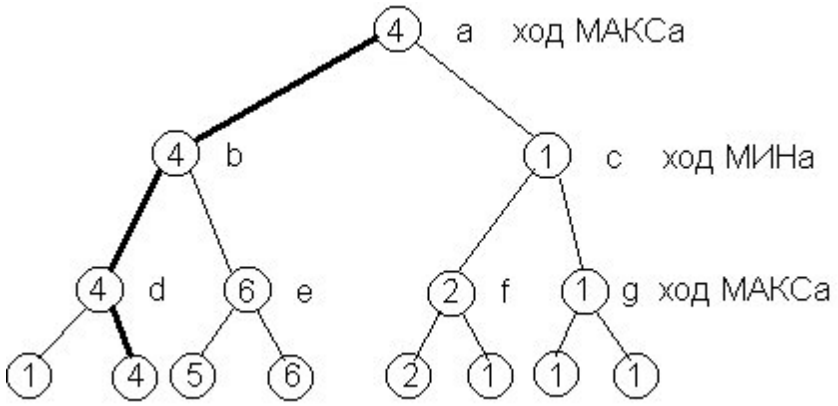


Рис. 2.11. Статический (нижний уровень) и минимаксные рабочие оценки вершин дерева поиска. Выделенные ходы образуют основной вариант, т. е. минимаксно-оптимальную игру с обеих сторон

Мы различаем два вида оценок: оценки вершин нижнего уровня и оценки внутренних вершин (рабочие оценки). Первые из них называют также "статическими", так как они вычисляются при помощи "статической" оценочной функции, в противоположность рабочим оценкам, получаемых "динамически" при распространении статических оценок вверх по дереву.

Правила распространения оценок можно сформулировать следующим образом. Будем обозначать статическую оценку позиции P через v(P), а ее рабочую оценку - через V(P). Пусть P_1, P_2, ... P_n - разрешенные преемники позиции P. Тогда соотношения между статическими и рабочими оценками можно записать так:

$V(P) = v(P)$

если P - терминальная вершина позиции дерева поиска (n=0);

$V(P) = \max_i V(P_i)$

если P - позиция с ходом МАКСа;

$V(P) = \min_i V(P_i)$

если P - позиция с ходом МИНа.

Приведем упрощенную программу на Прологе, вычисляющую минимаксную рабочую оценку для некоторой заданной позиции.

```
% Минимаксная процедура: minimax(P,GP,V)
% P - позиция, V - ее минимаксная оценка;
% лучший ход из позиции P ведет в позицию GP

minimax(P,GP,V):-
      'ходы'(P,List),!,            % List - список разрешенных ходов
      'лучшая'(List,GP,V);
     'статическая оценка'(P,V).  % позиция P не имеет преемников

'лучшая'([P],P,V):-
     minimax(P,_,V),!.
'лучшая'([P1|List],GP,GV):-
     minimax(P1,_,V1),
```

```
    'лучшая'(List,P2,V2),
    'выбор'(P1,V1,P2,V2,GP,GV).

'выбор'(P0,V0,P1,V1,P0,V0):-
    'ход МИНа'(P0),V0>V1,!;
    'ход МАКСа'(P0),V0<V1,!.
'выбор'(P0,V0,P1,V1,P1,V1).
```

Основное отношение этой программы minimax(P,GP,V), где V - минимаксная оценка позиции P, а GP - наилучшая позиция-преемник позиции P (лучший ход, позволяющий достигнуть оценки V). Предикат 'ходы'(P,List) задает разрешенные ходы игры: List - это список разрешенных позиций-преемников позиции P. Предполагается, что цель 'ходы' имеет неуспех, если P является терминальной поисковой вершиной (листом дерева поиска). Отношение 'лучшая'(List,GP,V) выбирает из списка позиций-кандидатов List "наилучшую" позицию GP. V - оценка позиции GP, а, следовательно, и позиции-предка. Под "наилучшей" оценкой мы понимаем либо максимальную, либо минимальную оценку, в зависимости от того, с чьей стороны ожидается ход.