МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «РЯЗАНСКИЙ ГОСУДАРСТВЕННЫЙ РАДИОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ИМЕНИ В.Ф. УТКИНА»

Кафедра «Вычислительная и прикладная математика»

ОЦЕНОЧНЫЕ МАТЕРИАЛЫ ПО ДИСЦИПЛИНЕ «Проектирование интеллектуальных автоматизированных систем»

Специальность

09.05.01 «Применение и эксплуатация автоматизированных систем специального назначения»

Специализация

«Математическое, программное и информационное обеспечение вычислительной техники и автоматизированных систем»

Уровень подготовки – специалитет

Квалификация выпускника – инженер

Форма обучения – очная

Срок обучения – 5 лет

Рязань

ВВЕДЕНИЕ

В результате развития средств электронной техники довольно быстро стало ясно, что собственно математические вычисления не являются единственными решаемыми с помощью ЭВМ задачами. Появились работы ученых, позже отнесенные к новой науке с названием “кибернетика”, введенным Н. Винером. Эти работы лавинообразно вызывали к жизни крупные классы программных систем, имеющих точно определенные задачи, теоретические методы их решения и способы реализации средствами вычислительной техники. Такими системами, например, явились информационно-поисковые системы (ИПС), автоматизированные системы управления (АСУ), системы автоматизированного проектирования (САПР), автоматизированные обучающие системы (АОС) и многие другие.

Развитие сложных программных систем потребовало развития базы средств вычислительной техники, ориентированной не просто на переработку, но и на хранение информации. Стало понятно, что данные вычислительных машин значительно чаще имеют не числовую, а информационную природу. Возникли понятия баз данных и средств удаленного доступа к информационным массивам. Само понятие электронно-вычислительной машины стало чаще употребляться не в форме аббревиатуры, а скорее как имя нарицательное. Современная научно-техническая литература чаще использует понятие “компьютер”, предполагающее новый, информационный взгляд на ЭВМ, т.е. весьма удаленное от дословного английского перевода “вычислитель”.

Искусственный интеллект как наука развивался практически одновременно с развитием языков программирования. Основная задача, которая положена в основу исследований по искусственному интеллекту, - моделирование мыслительных способностей (деятельности) человека, была настолько фантастична и заманчива для исследователей, что сразу привлекла внимание великого множества талантливых ученых. Из ряда разрозненных теоретических концепций и практических программных реализаций наука об искусственном интеллекте быстро, как ни одна другая, сформировалась по своей методологической, понятийной и технологической основе. Реальные практические результаты, полученные в рамках теории распознавания образов, проектирования систем общения на естественном языке (системы контроля лексики и синтаксиса, машинный перевод и т.п.), экспертных и расчетно-логических систем, систем эвристического программирования показали, что искусственный интеллект как наука дает возможность проектировать особый класс программ - программные системы искусственного интеллекта.

Проектирование интеллектуальных программных систем сейчас является едва ли не самым перспективным направлением для приложения усилий разработчиков программ. Для получения оригинальных методов зачастую нет необходимости использовать сверхсовременные вычислительные ресурсы, а эффект от применения этих методов может оказаться настолько высоким, что тираж продаваемого интеллектуального программного обеспечения сможет принести даже сверхприбыль. Если автоматизированные программные системы так или иначе уже насытили рынок программного обеспечения в самых разных направлениях (научные, бухгалтерские, конструкторские и технологические программы), то в области интеллектуальных программ ближайшее десятилетие едва ли сможет выявить как преимущества каких-либо классов интеллектуальных систем, так и фирму-лидера в производстве таких программных средств. Кроме того, вполне возможно, что получение хороших теоретических методов позволит выйти на рынок программного обеспечения даже программистам-одиночкам.

Настоящее пособие в силу ограниченности объема не претендует на сколько-нибудь полное освещение методологии проектирования интеллектуальных программ, но вместе с тем, кроме начального познавательного материала, содержит описание оригинального подхода к проектированию и формальному исследованию принципиально новых интеллектуальных программных систем.

Г Л А В А 1

ПРОЕКТИРОВАНИЕ ИНТЕЛЛЕКТУАЛЬНЫХ РЕШАТЕЛЕЙ НА ОСНОВЕ ЭВРИСТИЧЕСКОГО ПРОГРАММИРОВАНИЯ

1.1. Понятие эвристического программирования

Интеллектуальные решатели задач были исторически одними из первых реальных программных систем, в которых использовались методы искусственного интеллекта. Первые реализации игровых программ, таких как рэндзю, шашки и шахматы, показали состоятельность теоретических концепций искусственного

интеллекта и вызвали к жизни множество новых проектов, направленных на дальнейшее развитие интеллектуальных программных систем.

Основой интеллектуальных решателей задач, которые проектировались для моделирования мыслительной деятельности человека, были различные алгоритмические стратегии поиска решения в сложных предметных областях, основанные на так называемых эвристиках. Само понятие эвристики (в переводе с греческого означает способствование открытию) характеризует разновидность однотипных программных процедур, не обладающих строгими свойствами алгоритма и направленными на поиск решения посредством проверки различных по степени обоснованности гипотез. Таким образом, эвристические процедуры всегда связаны с работой некоторого переборного механизма, предназначенного для исследования вариантов решения. Проектирование таких процедур получило название эвристического программирования. В последние годы, как это часто бывает с наиболее серьезными теоретическими концепциями, в понятие эвристического программирования или эвристических систем стали вкладывать более широкий смысл. Сейчас такими системами считаются программные комплексы, использующие неформализованные знания эксперта в какой-либо конкретной предметной области для решения интеллектуальных задач. Такое понимание эвристических систем иногда может быть оправдано, но, вместе с тем, следует отличать эти системы от экспертных систем, в которых основой являются не стратегии поиска решения, а собственно знания, представляемые в форме упорядоченного множества продукций (правил вида “если X, то Y”) или каким-либо другим образом.

Современное понимание эвристических систем может быть наиболее точно выражено с помощью понятия программной машины [1], как некоторого формального аппарата, позволяющего описать базовый программный или программно-технический инструментарий как формальную машину, в рамках которой можно проектировать различные по назначению, но одинаковые по методам реализации автоматизированные системы.

Эвристические программные машины проектируются со следующими целями:

- упрощение синтаксиса интеллектуальных языков программирования и получение новых более мощных инструментальных программных средств,
- исследование структуры естественного интеллекта,
- создание спецпроцессора на основе предварительной математически строгой разработки соответствующей программной машины.

Эвристическое программирование применимо там, где невозможно использовать готовый математический аппарат, обладающей возможностью с заданной точностью, достоверностью и эффективностью решать определенного типа задачи с многовариантным полем шагов решения. Такое определение области применения эвристических методов в целом характерно и для других методов искусственного интеллекта, поскольку эти методы построены на подобии человеческим рассуждениям со свойственными им интуицией, ассоциативностью и нечеткостью.

1.2. Деревья вариантов

Формальную основу методов эвристического программирования составляет понятие дерева вариантов.

Дерево вариантов - дерево, вершины которого соответствуют статическим ситуациям, получаемым в математической модели предметной области при попытках рассмотрения вариантов изменения этих ситуаций для приближения их к целевой ситуации предметной области. Изменение ситуации соответствует некоторому действию человека, который хочет приблизить ситуацию к целевой. Такие изменения иногда называют “ходами”, которые оценивает эвристическая программа, после чего выбирает наилучшую комбинацию ходов из какого-либо множества вариантов.

Целевая ситуация - математическая модель ситуации в ограниченной предметной области, которая должна быть достигнута для решения поставленной задачи. Примером целевой ситуации может быть, например, матовая ситуация в шахматах, равновесное состояние некоторого химического процесса, отсутствие аварийного ожидания, получение высокой прибыли предприятием и т.п. .

Деревья вариантов принято подразделять на эксплицитные и имплицитные.

Эксплицитным называется дерево вариантов, все вершины которого заранее известны. Такое дерево, например, можно использовать при компьютерном моделировании игры в “крестики-нолики” на поле размером 3x3, поскольку в этом случае легко перечислить все возможные варианты ходов заранее. Для более сложных задач невозможно заранее определить всего множества ходов, и этот случай является наиболее частым.

Имплицитным называется дерево вариантов с известной начальной ситуацией в предметной области и известными критериями определения целевой ситуации. Кроме того предполагается, что известна порождающая процедура.

Порождающая процедура - алгоритм порождения дочерней вершины (на основе выполнения хода) в дереве вариантов.

Таким образом имплицитное дерево строится программой в процессе поиска ею решения интеллектуальной задачи.

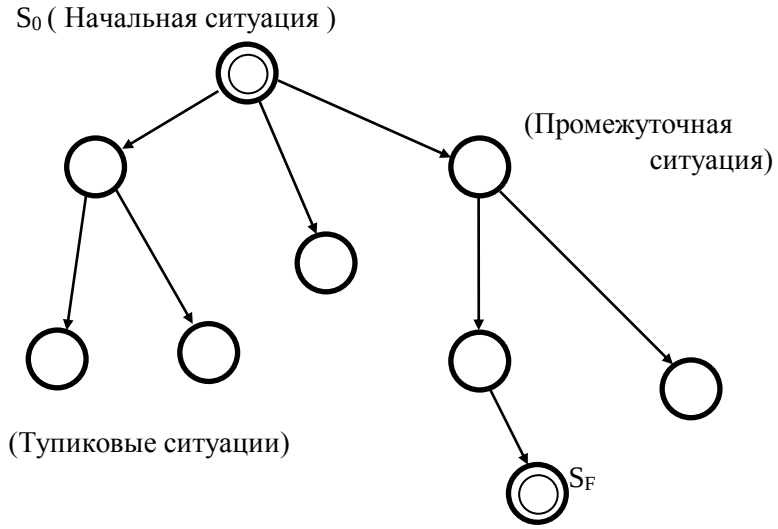
Сделанные определения позволяют рассматривать деревья вариантов как множество вершин, связанным между собою ребрами. Каждая вершина соответствует некоторой воображаемой ситуации в предметной области, а выходящие из нее ребра представляют собой возможные шаги, изменяющие эту ситуацию, приближая ее к целевой или удаляя от цели. Предполагается, что изменение ситуации производится некоторым активным объектом (например, игроком) по определенным правилам, допускаемым природой предметной области. Ребра, выходящие из какой-либо вершины, можно называть ходами, возможными в конкретной ситуации предметной области, которой соответствует эта вершина. Дерево вариантов демонстрируется рисунком 1.1 .

На этом рисунке вершина S_0 соответствует начальной ситуации, заданной для решения какой-либо задачи в предметной области, например, начальная расстановка фигур в шахматах. S_F соответствует целевой ситуации, например, ситуации в задаче поиска выхода из лабиринта “найден выход”. Остальные вершины являются либо промежуточными ситуациями при решении задачи, либо заключительными-тупиковыми (например, робот в лабиринте наткнулся на стенку или обнаружена ситуация полного проигрыша).

При исследовании деревьев вариантов учитывают число активных объектов в предметной области, которые могут изменять в ней ситуацию, приближая ее к целевой или наоборот, удаляя от целевой. Каждый из активных объектов таким образом является “игроком”, делающим определенные правилами предметной области “ходы”. У игроков различные цели в игре, зачастую являющиеся противоположными целям других игроков.

В зависимости от числа активных объектов деревья вариантов подразделяют на *однородные*, т.е. с одним активным объектом, и *неоднородные*, с двумя и более активными объектами. Графически такие деревья можно представить рисунком 1.1 для однородного дерева и рисунком 1.2 для неоднородного дерева вариантов.

Каждая из вершин этого дерева должна быть как-либо идентифицирована с целью определения объекта, который повлиял на самое последнее изменение текущей ситуации.



(Целевая ситуация)

Рис. 1.1. Пример дерева вариантов

Так, например, дерево вариантов для игры в шахматы является неоднородным с двумя типами вершин, одни из которых соответствуют ходу черных, а другие - ходу белых.

1.3. Пространство задач и пространство состояний

В зависимости от способа представления задачи и способов ее решения различают деревья поиска:

* дерево поиска в пространстве состояний,
* дерево поиска в пространстве задач.

Дерево поиска в пространстве состояний - это дерево, вершинами которого являются состояния решающей машины (программы) (Рисунки 1.1, 1.2), т.е. текущая ситуация на используемой модели решения, а также текущее содержимое памяти компьютера для выбранной модели данных.

Дерево поиска в пространстве задач - дерево, вершины которого являются подзадачами, решение которых необходимо для решения задач более верхнего уровня.

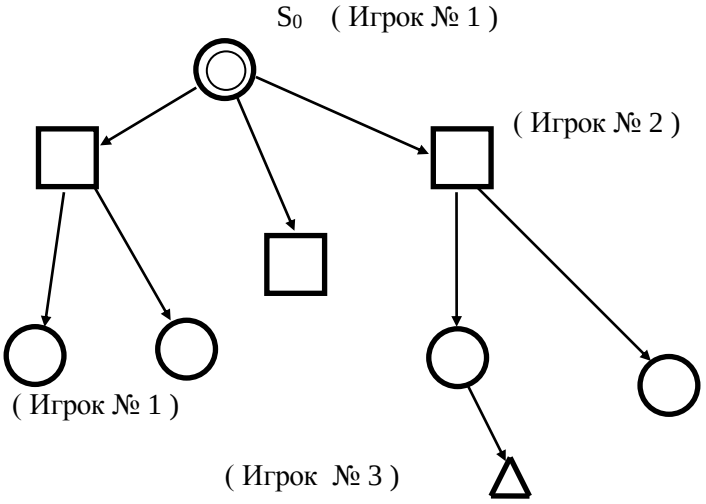


Рис. 1.2. Неоднородное дерево вариантов

Для дерева поиска в пространстве задач, иногда называемого *деревом целей*, характерно использование И/ИЛИ-графов. Этот вид графов имеет два типа вершин: И-вершины, характеризующие задачу, для решения которой необходимо отыскать решения всех задач, которые соответствуют дочерним вершинам И-вершины, ИЛИ-вершины соответствуют задачам, для решения которых необходимо найти решение хотя бы одной из подзадач, которые соответствуют дочерним вершинам этой ИЛИ-вершины.

Примером И/ИЛИ-деревьев служит представление задачи, приведенное на рисунке 1.3.

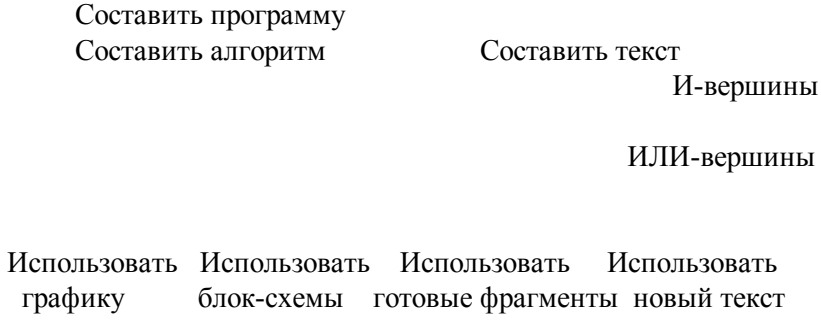


Рис.1.3. Пример И/ИЛИ-дерева

Как следует из рисунка, начальной задачей является достижение цели“Составить программу”. Эта задача распадается на две подзадачи, решение которых нужно получить для решения главной задачи. Эти подзадачи определяются И-вершинами, т.е. должны быть решены вместе: “Составить алгоритм” и “Составить текст”. Для решения каждой из задач этого уровня необходимо найти решение хотя бы одной из подзадач: “Составить алгоритм, используя графический инструментарий” или “Составить алгоритм, используя ручное рисование блок-схем” - в первом случае, и “Написать текст, используя готовые фрагменты” или “Написать текст программы заново”.

Если продолжить рисование И/ИЛИ-дерева, далее будет следовать уровень И-вершин, затем снова ИЛИ-вершины и т.д., т.е. чередование типов вершин по уровням является строгим. Эта строгость определяется элементарной математической логикой, следуя которой, два последовательных уровня И-вершин легко преобразуются в один уровень, также, как два уровня ИЛИ-вершин легко преобразовать в один. Методология И/ИЛИ деревьев часто используется в программных машинах языков программирования искусственного интеллекта, например в Прологе.

Контрольные вопросы к главе1

1. В чем состоит понятие “эвристическое программирование”?
2. Перечислите цели использования методов эвристического программирования.
3. Каковы основные понятия теории эвристического программирования?
4. Что такое “дерево вариантов”?
5. Используется ли эвристическое программирование для задач, в которых получены некоторые математические решения?
6. Чем отличаются эксплицитные и имплицитные деревья вариантов?
7. Как называются ситуации, которые могут составлять основу деревьев вариантов?
8. Приведите отличия методов поиска в пространстве состояний и в пространстве задач.
9. На каком из методов поиска построен язык программирования Пролог?
10. Чем отличаются вершины в деревьях вариантов типа “И” от вершин типа “ИЛИ”?
11. Попробуйте привести примеры задач для поиска в пространстве состояний и в пространстве задач.
12. В чем отличие однородных и неоднородных деревьев вариантов? Нарисуйте какие-либо примеры этих видов деревьев.
13. Что представляют собой вершины дерева вариантов при поиске в пространстве задач?
14. Что представляют собой вершины дерева вариантов при поиске в пространстве состояний?
15. Чему соответствует число дочерних вершин какой-либо вершины в дереве вариантов?
16. Что такое порождающая процедура?
17. Что может пониматься под понятием тупиковой ситуации?
18. Дайте определение целевой ситуации.
19. Что понимается под понятием “игрок” в эвристическом программировании? Какое максимальное число игроков может быть задействовано в задачах эвристического программирования?

Г Л А В А 2
ПОИСК НА ДЕРЕВЬЯХ

2.1. Поиск на деревьях вариантов

К базовым эвристическим процедурам относят переборные алгоритмы поиска на деревьях: поиск в глубину и поиск в ширину.

Поиск в глубину основан на последовательном переборе вариантов изменения начальной ситуации начиная каждый раз с самого левого потомка вершины. При этом исследуются дочерние вершины дерева поиска до обнаружения решения, т.е. целевой вершины или тупика. В случае обнаружения тупика поисковая процедура реализует возврат к отцовской (порождающей) вершине и осуществляет попытку рассмотрения очередного варианта. Алгоритм заканчивается при отыскании любой из целевых вершин или при обнаружении ситуации, в которой начальная вершина является тупиковой.

Тупиковой называется вершина, относящаяся к тупиковым соответствующим предикатом или вершина, все потомки (дочерние вершины) которой - тупиковые.

Поиск в ширину основан на переборе вариантов параллельно во всех возможных направлениях на дереве поиска. Этот алгоритм исследует сначала все вершины уровня 1, затем все вершины уровня 2, начина от корневой вершины, и т.д. до тех пор, пока не будут рассмотрены все варианты до ближайшего решения (целевой вершины) или будут обнаружены все тупиковые вершины.

Рассмотренные два метода можно сравнить. Так, например, поиск в ширину пригоден в тех случаях, когда разветвленность дерева вариантов достаточно мала. Под *разветвленностью* понимается среднее число ветвей-потомков, выходящих из одной вершины дерева вариантов. Вместе с тем, поиск в ширину позволяет отыскивать оптимальное решение, т.е. самый короткий путь от начальной вершины дерева вариантов до целевой вершины.

При большой разветвленности дерева вариантов поиск в ширину малоэффективен по соображениям затрат времени и памяти. В этом случае применяют поиск в глубину, который отыскивает первое приемлемое (неоптимальное) решение.

При ограничении числа уровней просмотра дерева вариантов оценки эффективности обоих методов тем более близки друг к другу, чем меньшее число уровней просмотра дерева вариантов используется. Если перебор в глубину ограничивать всякий раз единственным уровнем просмотра вариантов, то этот метод превращается в метод поиска в ширину.

Оба рассмотренных метода считаются базовыми методами эвристического программирования, поскольку рассчитаны на тривиальный вариант начальный знаний о предметной области, т.е. считается, что процедура поиска не обладает возможностью оценивания наиболее лучших вариантов.

Более совершенными являются методы поиска, предполагающие наличие априорных знаний о предметной области. Эти знания составляют основу оценочных функций. *Оценочной функцией* называется функция от n аргументов, представляющих собой некоторые числовые (или приведенные к таковым) параметры предметной области, и вычисляющая меру близости любой из текущих ситуаций в модели предметной области к целевой ситуации.

Значения оценочной функции представляют собой точки n-мерного пространства. Множество таких точек может быть представлено поверхностью в этом пространстве. Функция оценки дает возможность выбирать направления поиска в дереве вариантов для более быстрого отыскания целевой вершины (ситуации). Эта функция в некоторых случаях не может быть эффективно построенной, например, при полном отсутствии каких-либо знаний о предметной области. Существует, однако, критерий, позволяющий совершенно уверенно определить, можно ли построить оценочную функцию или нет. Он заключается в наличии четко заданных начальной и заключительной ситуаций. Если их описание достаточно конструктивно, то на математической модели ситуаций можно вычислить различие между целевой ситуацией и какой-либо текущей по некоторому множеству числовых параметров. Это и будет оценочной функцией.

Примером оценочной функции может быть определение расстояния от текущего положения робота до выхода его из лабиринта в задаче поиска пути в лабиринте. Для игры рэндзю (“крестики-нолики”) эта функция может вычисляться следующим выражением:

$$f = 1 * (x_1 - y_1) + 2 * (x_2 - y_2) + 3 * (x_3 - y_3) + 4 * (x_4 - y_4) ,$$

где $x_i - y_i$ - разность стоящих в один ряд камней игрока и противника, число которых равно i. Этот пример говорит о необходимости вычисления оценочной функции в игровых задачах как разности характеристик выигрышности позиций соперников. Если соперников несколько, необходимо оценивать выигрыш каждого с каждым.

2.2. Стратегии поиска с оценочными функциями

Метод наискорейшего подъема (поиск по градиенту)

В реальной предметной области локальные измерения при помощи датчиков (или функций оценки) превращаю поиск в ширину в метод наискорейшего подъема (поиска по градиенту).

Метод заключается в правиле максимизации. Необходимо сделать один шаг по дереву вариантов в каждом возможном направлении. Затем переместиться в самую лучшую вершину, судя по оценочной функции и повторять это действие до отыскания целевой вершины. Иными словами, при поиске решения этим методом каждый новый ход выполняется после оценки всех возможных в конкретной ситуации ходов. Выбирается каждый раз ход, для которого оценочная функция дала наилучшую оценку. Наилучшая оценка

может быть максимум или минимумом оценочной функции в зависимости от того, как составил эту функцию сам программист.

Вполне естественно, этот метод не гарантирует отыскания оптимального пути до целевой ситуации. Мало того, воспользовавшись им во многих задачах можно не найти решение даже в том случае, если оно реально существует. Действительно, выбирая каждый раз самый лучший вариант в дереве вариантов, мы оставляем нерассмотренными все остальные, которые вполне могут вести к целевой ситуации.

Для избежания указанного недостатка базовый метод поиска по градиенту преобразуют в метод, смешанный с перебором в глубину. При отыскании всех тупиковых ситуаций для какой-либо вершины пытаются проверить следующий вариант, который был оценен ранее как менее предпочтительный.

Метод наискорейшего подъема будет еще более результативен, если задаться некоторым ограничением просмотра, например, на n шагов. В этом случае, если за n шагов не обнаружится решение, можно вернуться к рассмотрению ранее нерассмотренных вариантов.

Поиск от наилучшего частичного пути

Усовершенствование метода наискорейшего подъема заключается в рассмотрении не одного уровня в дереве вариантов, а сразу нескольких с последующей оценкой частичного пути, т.е. каждой из рассмотренных в дереве вариантов ветвей. За оценку принимают либо сумму всех оценок пути, либо среднее арифметическое. При этом, в отличие от предыдущего метода, выбираются несколько наиболее лучших вершин, и поиск продолжается, начиная с них (с возможным возвратом к еще нерассмотренным вершинам).

Стратегия ветвей и границ

Эта стратегия использует оценочную функцию, представляющую собой функцию вычисления сложности выполнения шага в дереве вариантов. Она, например, может соответствовать в реальной предметной области длине пройденного пути, который должен быть по возможности короче.

Стратегия заключается в расчете оценок-весов первого уровня дерева вариантов и последующем расчете оценок следующих уровней. При продвижении по дереву выбирается ветвь с наименьшей оценкой суммарного частичного пути, что позволяет несколько сократить число рассматриваемых и оцениваемых вариантов с опорой на поиск в ширину.

Для понимания метода достаточно привести следующий пример (рис.2.1).

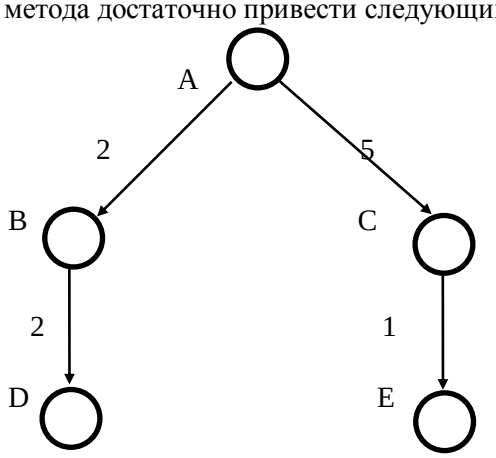


Рис. 2.1. Пример стратегии ветвей и границ

Оценку пути ветви CE в этом примере не имеет смысла производить, поскольку суммарный вес частичного пути ABD, равный 2 + 2 = 4, меньше, чем AC, равный 5. В этом случае имеет смысл исследовать далее варианты, начиная с вершины D. Такое исследование неизбежно приведет к тому, что суммарная оценка частичного пути ABD… превысит оценку пути AC и, следовательно, будет необходимо вернуться к рассмотрению пути CE.

Стратегия равных цен

Алгоритм равных цен характерен для задач с частым повторением ситуаций на модели предметной области при просмотре вариантов решения.

Характерным примером для этого метода поиска является задача о перевозках, в которой необходимо определить кратчайший проезд из начального пункта в целевой среди нескольких возможных вариантах такого проезда.

Проиллюстрируем этот метод следующим рисунком (рис.2.2).

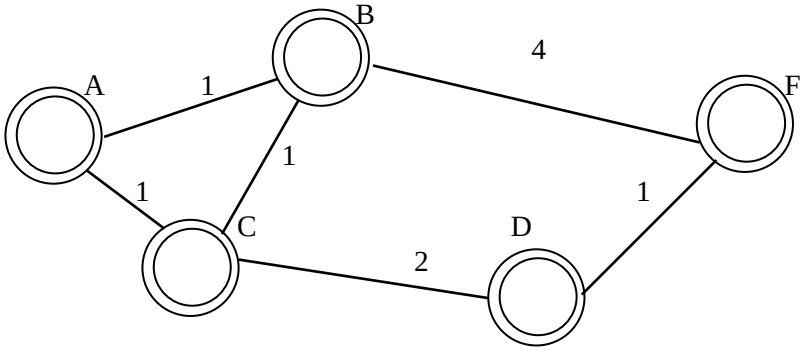


Рис. 2.2. Пример задачи о перевозках

В примере необходимо найти кратчайший путь из пункта A в пункт F.

Поиск при этой стратегии осуществляется на основе базовой эвристике перебора в ширину. При этом все достигнутые состояния (ситуации) получают оценки аналогично поиску по стратегии ветвей и границ, т.е. оценки суммарного частичного пути, пройденного до соответствующего состояния.

Для каждого из состояний запоминается кратчайший путь до него. С этой целью при достижении порождающей процедурой какого-либо состояния анализируется полученная ранее оценка пути до этого состояния, если состояние уже было достигнуто когда-либо ранее. При этом выбирается самая лучшая из оценок и запоминается путь, который привел в это состояние кратчайшим образом. В том случае, когда новое состояние достигнуто впервые, ему приписывается единственная полученная оценка, которая в последствии может быть изменена при отыскании более короткого пути.

Для приведенного на рис.2.2 примера алгоритм стратегии работает так. Оцениваются пути AB и AC с приписыванием состояниям B и C оценок, равных 1. Далее оцениваются одновременно пути CB, CD и BC, BF. При этом получается, что до состояния B отыскивается еще один путь длиной 2, что хуже предыдущего варианта и поэтому оценка состояние не меняется. Состояние D получает оценку суммарного частичного пути, равную 3, а состояние F, равную 5. Заключительным шагом является оценка пути DF, исправляющая ранее полученную для этого состояния оценку 5 на более хорошую оценку 4. Эта оценка и выявляет кратчайший путь из A в F как путь ACDF.

Алгоритм равных цен гарантирует отыскание оптимального пути в дереве вариантов.

Рассмотренные стратегии поиска, используемые в интеллектуальных решателях задач, могут различаться по направлению поиска на прямой и обратный.

При *прямом* поиске исследование вариантов ведется от начальной ситуации к целевой, при *обратном* - от целевой к начальной.

Существуют исследования, доказывающие предпочтительность комбинированного метода поиска, при котором варианты рассматриваются одновременно (поочередно) в двух направлениях.

2.3. Игровые стратегии поиска решений

На неоднородных деревьях вариантов (п.1.1) можно рассматривать *игровые стратегии поиска решения*, т.е. стратегии, предназначенные для выбора наилучшего “хода”, направленного на выигрыш у противника в какой-либо предметной области, называемой “игрой”. В дальнейшем изложении будем опускать кавычки у слов “ход” и “игра”, поскольку основные примеры игровых стратегий удобнее всего рассматривать для интеллектуальных игр, таких как шахматы, шашки, карточные и другие игры.

Минимаксный подход в игровых стратегиях

Для двух игроков можно использовать так называемый *минимаксный* подход к поиску наилучшего текущего хода.

В этом случае интеллектуальной программой используется одна и та же оценочная функция для выбора предпочтительного хода как для одного, так и для другого игрока. Это определяется тем соображением, что хороший ход предполагает необходимость рассмотрения как нескольких вариантов “своих” ходов, так и возможные ответы противника.

Поскольку компьютер, играющий против человека или другого компьютера, не может знать какими соображениями руководствуется его противник, т.е. его интеллектуальный уровень и сумму знаний в игровой предметной области, то приходится предполагать, что противник, по крайней мере также силен, как и играющая с ним программа. Следовательно, оценивая его возможные ходы, нужно использовать свою собственную оценочную функцию от имени противника, поскольку более мощной функцией программа просто не располагает.

Оценочная функция для одного игрока в минимаксном подходе вычисляется со знаком “+”, и считается, что игрок должен выбрать ход с максимальным значением этой функции, для другого эта функция вычисляется со знаком “-“, и выбор хода при этом определяется минимальностью значения функции. В соответствии с такими вычислением функций оценки одного игрока принято называть “макс”, а другого “мин”.

Таким образом каждый из игроков при выборе варианта хода использует ход с наилучшей для него оценкой (мин - минимальной, макс - максимальной).

За счет сделанных предположений о выборе противником лучшего хода существенно сокращается число рассматриваемых возможных вариантов продолжения игры. Поиск становится более направленным.

Вполне естественно, что оценка текущей ситуации в игре тем более точна, чем больше глубина просмотра возможных взаимных ответных ходов (просмотр на большее число ходов вперед).

Выбор численного значения оценки осуществляется с самого глубокого уровня с постепенным подъемом по ребрам дерева вариантов к самой верхней вершине, характеризующей текущую ситуацию в игре.

Рассмотрим следующий рисунок (рис.2.3).

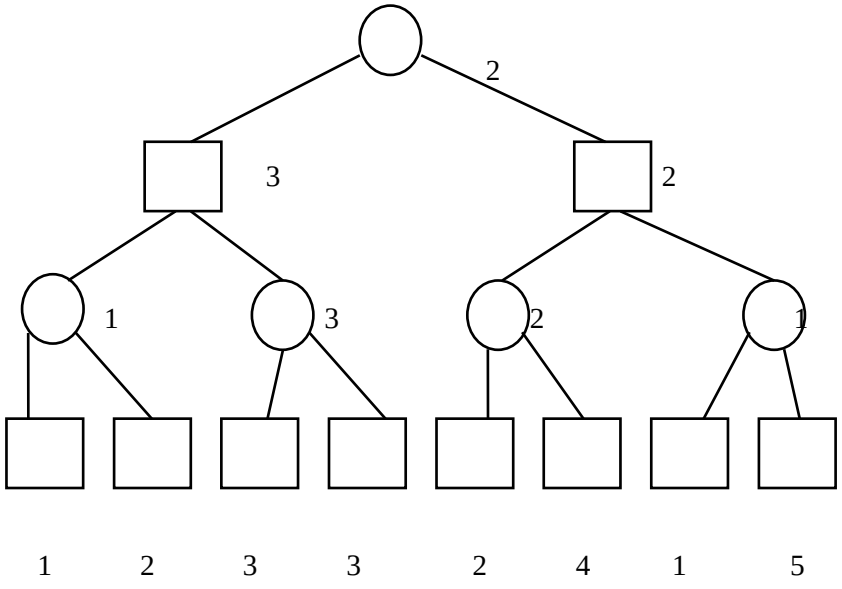


Рис.2.3. Пример минимаксного вычисления оценочной функции

Здесь оценки самого нижнего уровня получены какой-либо решающей процедурой, оценивающей игровую ситуации вообще. Все другие оценки получаются выбором минимальной или максимальной оценки более низкого уровня в зависимости от того, кем из игроков делается ход на этом уровне (мином или максом).

Минимаксный подход дает возможность использовать *метод локально-углубленного поиска решения*.

Метод локально-углубленного поиска

Этот метод заключается в ограничении просмотра ходов для оценки поиском в ширину некоторым числом уровней n. При этом выбирается лучший из рассмотренных ходов и осуществляется более глубокая его оценка (в глубину на m уровней). В том случае, если предыдущая оценка уточняется в нехудшую сторону, принимается решение о выполнении хода в реальной игре. В противном случае рассматриваются более подробно ходы, оцененные ранее как менее предпочтительные.

Усовершенствование метода локально-углубленного поиска носит название “метод α/β-отсечений”.

2.4. Метод α/β-отсечений

Метод заключается в отказе от просмотра вариантов ходов, на которые противник может ответить заведомо более сильным ходом, чем на ранее оцененные ходы.

Суть метода можно пояснить рисунками 2.4 и 2.5 .

α-отсечение используется обоими соперниками при просмотре ходов макс-игрока в том случае, если какой-либо из рассматриваемых вариантов приписывает порождающей вершине дерева вариантов оценку, заведомо меньшую, чем оценка предыдущей порождающей вершины.

β-отсечение используется обоими соперниками при просмотре ходов мин-игрока в том случае, если какой-либо из рассматриваемых вариантов приписывает порождающей вершине дерева вариантов оценку, заведомо большую, чем оценка предыдущих порождающих вершин.

В этом методе речь идет об оценке программой ходов сначала за макс-игрока, затем за мин-игрока. При этом одним из игроков является, по сути дела, сама программа.

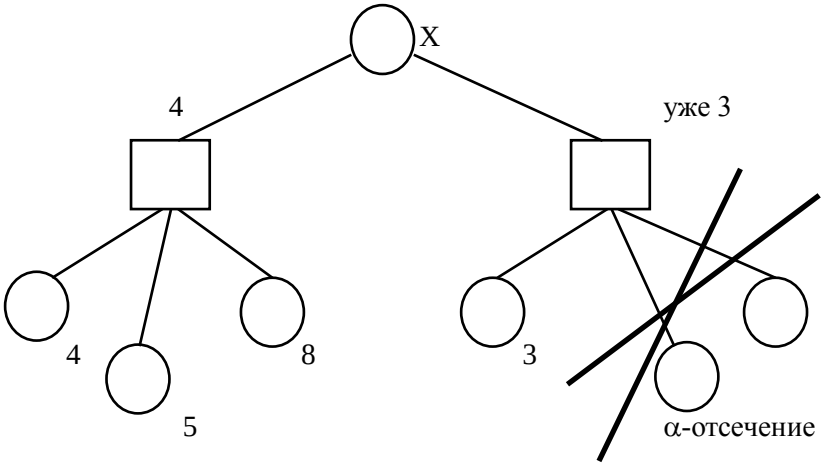


Рис. 2.4. α-отсечение

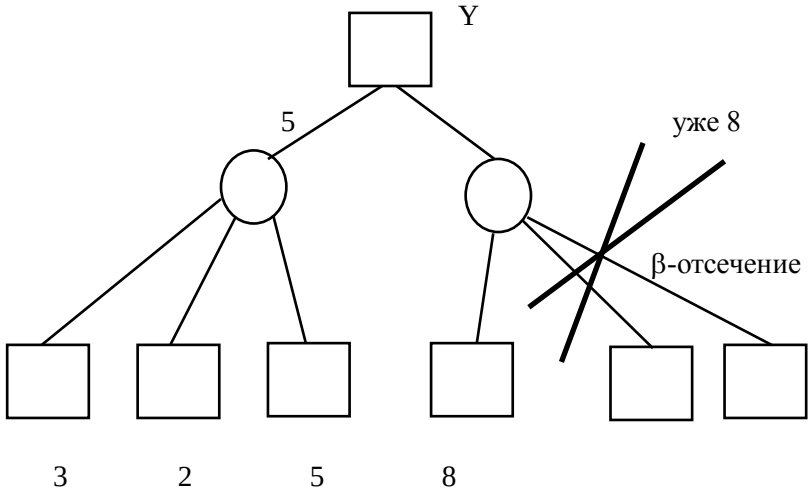


Рис. 2.5. β-отсечение

Контрольные вопросы к главе2

1. Какие методы поиска относят к базовым переборным алгоритмам?
2. В чем суть и отличие методов перебора в глубину и в ширину?
3. Дайте определение тупиковой вершины.
4. При каких условиях оканчиваются алгоритмы поиска на деревьях вариантов? Какими могут быть возможные результаты поиска?
5. Что такое “разветвленность дерева вариантов”?
6. Какой из базовых переборных методов позволяет найти оптимальное решение?
7. В каком случае лучше использовать поиск в ширину, а в каком - в глубину?
8. Что такое “оценочная функция”? Для чего она используется при поиске?
9. Приведите свой пример оценочной функции.
10. Для каких задач можно достоверно утверждать, что оценочная функция для них обязательно существует?
11. В чем недостатки метода наискорейшего подъема? Какие усовершенствования этого метода существуют ?
12. Сравните методы *равных цен* и *ветвей и границ*. Какой из этих методов гарантирует отыскание оптимального пути?
13. Что такое “прямой” и “обратный” поиски?
14. Какая модификация прямого и обратного поисков приводит к повышению эффективности поиска?
15. Из какого предположения об интеллектуальной силе противника исходит идея минимаксного подхода к оценке текущей ситуации игры?
16. Приведите пример расчета оценок по минимаксному методу для произвольно выбранного дерева вариантов.
17. На чем основан метод локально-углубленного поиска решения, использующий минимаксный подход?
18. В чем отличие α- отсечения от β-отсечения? Какое из этих отсечений используется программой при просмотре ходов мин-игрока?

Г Л А В А 3

ПРОГРАММНАЯ РЕАЛИЗАЦИЯ ИНТЕЛЛЕКТУАЛЬНЫХ РЕШАТЕЛЕЙ ЗАДАЧ

3.1. Циклическая и рекурсивная реализации интеллектуальных решателей задач

Наиболее часто рассматриваются два варианта программной реализации эвристических программ:

* реализация с помощью циклических конструкций,
* реализация с помощью рекурсивных процедур.

Циклические конструкции позволяют более просто управлять механизмами возврата к более верхним уровням деревьев вариантов при обнаружении тупиковых вершин. При этом в программе достаточно задать переменную, содержащую номер уровня, и некоторый массив или список данных, описывающий ситуацию в предметной области (элемент модели данных). В то же время, при составлении программы необходимо позаботиться о выделении оперативной памяти для построения новых вершин дерева вариантов, а также своевременном ее освобождении при необходимости возврата к отцовским вершинам. К циклическим конструкциям можно отнести известные конструкции do-while или for универсальных языков программирования. Внешний цикл организуется по номеру уровня в дереве вариантов.

Рекурсивные алгоритмы отличаются максимальной простотой, поскольку автору интеллектуального решателя не приходится заботиться об отведении памяти под новые генерируемые процедурой порождения вершин ситуации в модели предметной области, равно как и об освобождении памяти, занятой тупиковыми вершинами. Работа с оперативной памятью компьютера в этом случае автоматически выполняется при рекурсивном вызове процедуры генерации новой вершины дерева вариантов.

Упрощенно рекурсивная процедура выглядит следующим образом.

1) Установить номер уровня дерева вариантов равным 0.

Главная часть рекурсивной процедуры

2) Увеличить номер уровня на единицу.

3) Если текущая ситуация является целевой, перейти к пункту 8, иначе выполнить следующий пункт.

4) Если текущая ситуация является тупиковой, перейти к 7 , иначе выполнить следующий пункт.

5) Сгенерировать дочернюю вершину дерева вариантов (новую ситуацию на модели предметной области).

6) Вызвать *рекурсивную процедуру* для обработки новой ситуации.

7) Если тупиковой является начальная (корневая) вершина дерева вариантов, завершить всю программу с выдачей отрицательного результата, иначе закончить процедуру, выдав результатом признак тупика.

8) Выдать текущий шаг решения, и, сформировав признак обнаружения целевой вершины, завершить процедуру.

Конец процедуры

3.2. Продукционный подход к реализации процедур поиска решения

Современные системы поиска решения интеллектуальных задач используются во многих видах прикладных автоматизированных программных систем. Наиболее известными из них являются:

- расчетно-логические системы,
- экспертные системы,
- логические машины выводы интеллектуальных языков программирования,
- системы ситуационного управления.

Объединение различных подходов в рамках этих систем к эвристическому поиску привело к необходимости использовать вместо классического понятия “ход” нового понятия “продукция” (или “правило”).

Продукцией (*правилом*) называется упорядоченная пара <условие применения продукции><действие продукции>. Под *условием применения продукции* понимается описание в программе ограничений, накладываемых на возможность выполнения каждого действия продукции (хода) в зависимости от текущей ситуации в предметной области (решаемой задаче). Например, в текущей точке лабиринта из непроходимых стен робот не может двигаться налево, поскольку там сейчас находится стена лабиринта. Под *действием продукции* понимается выполнение операций (хода) по преобразованию текущей ситуации, если условие применения продукции выполняется (истинно). Например, если слева от робота в лабиринте в текущей ситуации отсутствует стена, можно выполнить действие “шагнуть влево” (тем самым ситуация будет изменена).

Самой простейшей реализацией метода бесконечного перебора в глубину является использование неупорядоченного никакими оценочными функциями множества продукций для их применения к текущей ситуации. Каждый раз, когда необходимо выполнить ход (применить действие продукции), из этого множества выбирается первая попавшаяся продукция. Она применяется к текущей ситуации. Полученная в результате отработавшей продукции новая ситуация анализируется на наличие достижения цели и, если таковая не достигнута, процесс выбора продукций и их применения возобновляется.

Приведем схематическую запись алгоритма перебора в глубину на абстрактном универсальном языке программирования с использованием фраз естественного языка.

```
Program Simple_Heuristic ( Начальная ситуация S0 )
1. <Situations> ← S0
        {В область памяти, отведенную под множество}
        { возникающих в процессе поиска ситуаций      }
        { помещается начальная ситуация.              }
2. until  Goal (<Situations>)  do:
        {  Пока не будет достигнута целевая ситуация,  }
        {  распознаваемая логической функцией Goal,    }
        {выполнять следующую последовательность   }
        {операторов:}
3.     begin { Начало блока операторов. }
4.      <R> ← First ( <Rules> )
```

```
        {Выбрать из множества продукций <Rules>     }
        {первую, которую можно применить к          }
        {текущей ситуации, и поместить в область    }
        { памяти <R>.                                }
5.   <Situations> ← <R> (<Situations>)
        { В область <Situations> записать результат  }
        { применения <R> к предыдущей ситуации.      }
6.   end.{Конец блока операторов и всей программы.  }
```

Приведенный алгоритм описан весьма подробно в комментариях, выделенных фигурными скобками. Далее будем приводить реализации алгоритмов, в которых используются те же обозначения, в следствие чего они будут либо не комментироваться, либо комментироваться сокращенно.

Далее рассмотрим рекурсивную реализацию алгоритма поиска, в котором анализируются тупиковые ситуации и используется оценочная функция. Эта реализация более всего соответствует методу наискорейшего подъема (см.п.2.2).

```
Recursive Program Gradient_Heuristic ( Ситуация  S )
  1. if ( Goal (S) ), return (O'key); {Если цель, вернуть O'key. }
  2. if Deadend (S),return (FAIL);
                                        {Если тупик, вернуть "Неудача".}
  3. <Rules> ← Apprules (S); {  Apprules (Appreciate Rules) -   }
             { процедура упорядочивающая продукции }
             { с помощью оценочной функции и       }
             { выбирающая лучшую применимую к S    }
             { продукцию.                           }
  4. LOOP:     { Метка начала цикла.                 }
     if (NULL <Rules>) return (FAIL);
             { NULL - процедура проверки на пустоту.  }
  5. <R> ← First <Rules>;
  6. <Rules> ← Tail <Rules>; {Tail - процедура удаления     }
                             {отработавшей продукции.    }
  7. <RS> ← R (<S>);      {<RS> - новая область ситуаций. }
  8. <Result> ← Gradient_Heuristic ( RS );
 {<Result> - результат рекурсивного вызова программы.}
  9. if <Result> = FAIL, go LOOP;  { Переход на метку.       }
  10. Return Build(R, Result); {Здесь программа оказывается}
      {только при обнаружении цели. Build - процедура  }
     {составления решения из продукций, находящихся }
     {на пути решения.                               }
```

Заметим, что список применимых правил составляется в этой программе отдельно для каждой ситуации в дереве вариантов. Это дает возможность, вычеркивая уже выполненный ранее ход (<Rules> ← Tail <Rules>), при возврате из тупика выбирать очередной, еще не рассмотренный ход, до полного исчерпания всех возможных ходов (if (NULL <Rules>) return (FAIL)) для этой ситуации.

В то же время приведенная программа может попасть в бесконечную рекурсию, если в решаемой задаче через какое-либо число ходов ситуация повторится, поскольку в программе нигде не запоминаются уже рассмотренные промежуточные ситуации.

Приведем более эффективную программу, исключающую повторы ситуаций за счет хранения всего списка промежуточных нетупиковых ситуаций. Эта программа замечательна еще и тем, что в ней оптимально реализуется ограничение просмотра, определяемое исчерпанием памяти компьютера (в программе параметр BOUND - граница памяти), выделяемой под хранение промежуточных ситуаций.

Программа является рекурсивной. Ее входом является не одна ситуации, как это было в прежней программе, а список ситуаций. При старте программы список состоит из единственной начальной ситуации, а

при рекурсивных вызовах - из всех промежуточных, но первым в списке всегда находится последняя ситуация, только что полученная при выполнении предыдущего хода.

```
Recursive Program Intelligent_Heuristic(Ситуации SList )
1.  <S> ← First ( SList ){Выбор первой ситуации списка.    }
2.  if Member(S, Tail(SList)) return (FAIL);
                {Member - процедура проверки на вхождение }
                { новой ситуации S в ранее рассмотренные.   }
3.  if ( Goal (S) ), return (O'key); {Если цель, вернуть O'key. }
4. if Deadend (S),return (FAIL);
                                                {Если тупик, вернуть "Неудача".}
5.  if Length (SList) > BOUND, return(FAIL);
                        {Length - вычисление длины списка.}
6. <Rules> ← Apprules (S); {  Apprules (Appreciate Rules) -  }
                { процедура упорядочивающая продукции }
                { с помощью оценочной функции и          }
                { выбирающая лучшую применимую к S    }
                { продукцию.                                }
7. LOOP:        { Метка начала цикла.                    }
      if (NULL <Rules>) return (FAIL);
                { NULL - процедура проверки на пустоту.  }
8. <R> ← First <Rules>;
9. <Rules> ← Tail <Rules>; {Tail - процедура удаления      }
                        {отработавшей продукции.    }
10.<RS>← R(<S>);          {<RS>-новая область ситуации.}
11. <RSList> ← Build(RS, SList);
                  {Построение нового списка, в котором  }
                {первой будет новая полученная ситуация.}
12. <Result> ← Intelligent_Heuristic ( RSList );
  {<Result> - результат рекурсивного вызова программы.}
13. if <Result> = FAIL, go LOOP;  { Переход на метку.        }
14.Return Build(R, Result); {Здесь программа оказывается}
       {только при обнаружении цели. Build - процедура  }
       {составления решения из продукций, находящихся }
       {на пути решения.                                }
```

Замечательная особенность этой реализации в том, что она запоминает не все промежуточные ситуации, так как позволяет "забывать" те ситуации, которые лежали на тупиковых путях в дереве вариантов. При достаточной памяти программа гарантирует отыскание решения.

3.3. Оценка эффективности поиска

Для того, чтобы сравнивать методы и алгоритмы эвристического поиска между собой с целью выявления наиболее эффективных по затратам основных компьютерных ресурсов, рассчитывают основные *характеристики поиска*.

При этом исследуют следующие известные после работы программных реализаций составляющие:

- время поиска,
- объем занятой оперативной памяти,
- активизированное поддерево вариантов.

Активизированное поддерево поиска - это дерево, построенное имплицитно (см.п.1.2) в процессе анализа вариантов решения программой.

Все характеристики поиска разделяют на три класса: *базовые характеристики*, получаемые без каких-либо расчетов, *производные*, получаемые путем расчета их на основе знания базовых характеристик, и *эффективные характеристики*, рассчитываемые из характеристик первых двух классов с соотнесением их ко времени поиска.

Базовые характеристики поиска

1. *Глубина поиска* D: максимальное число вершин, порожденных до целевой или тупиковой вершины в дереве вариантов.
2. *Длина пути решения* L: число вершин, лежащих на минимальном из найденных путей решения.
3. *Общее число порожденных вершин* N.

Производные характеристики поиска

4) *Разветвленность поиска* $R = N / L$.

5) *Направленность поиска* $W = N^{1/L}$.

Для конкретных программ, реализованных на ЭВМ, можно определить также *эффективные* характеристики поиска.

6) *Максимальное хранимое поддерево* поиска M. Характеристика представляет собой максимальное число вершин, одновременно присутствующих в оперативной памяти как элементы соответствующей программной модели ситуаций решения задачи.

7) *Эффективность просмотра вершин* $t_c = T / N$, где T - время работы программы.

8) *Емкость поиска* $V_s = V_b * M$, где V_b - объем оперативной памяти, занимаемый моделью данных в байтах.

Эффективные характеристики поиска

Если характеристики 1- 5 отнести ко времени решения задачи, то можно получить соответствующие эффективные характеристики, например, D/T - *эффективная глубина поиска*, L/T - *эффективная длина пути решения* и т.д. .

Перечисленные характеристики дают возможность весьма достоверно оценить эффективность эвристических алгоритмов. При этом основой такой оценки является анализ разветвленности дерева поиска, просмотренного алгоритмом. Разветвленность представляет собой среднее число вершин потомков для каждой из вершин дерева вариантов. Это, как правило, действительное нецелое число.

Наилучшим алгоритмом поиска решения, следуя этой оценке, будет являться алгоритм, для которого $R = W = 1$, т.е. такой, для которого дерево поиска превращается в последовательный список вершин, представляющих собой промежуточные ситуации в предметной области на пути решения. В общем случае, такого алгоритма не может существовать для любой задачи, поскольку в этом случае предполагается, что решение каждой задачи заведомо известно.

Вместе с тем, при достаточно хорошей оценочной функции в алгоритме поиска можно добиться характеристик R и W, близкой к единице, поскольку абсолютно точная оценочная функция также превращает дерево вариантов в линейный список. В то же время, необходимо учитывать тот факт, что при абсолютном отсутствии знаний о предметной области решаемой задачи перебор, как составляющая алгоритма поиска, является необходимым.

При оценке работы реальных алгоритмов поиска используют выборку разнообразных задач. При определении тестирующего множества задач варьируют следующие параметры:

1) среднее число продукций (ходов), применимых к одной текущей ситуации,

2) емкость модели данных предметной области (число объектов, используемых при описании ситуаций),

3) сложность задачи (длина пути решения),

4) эффективность порождения.

После проведения серии экспериментов полученные оценки усредняют.

Если оценивается какой-либо новый алгоритм, и при этом для различных классов задач получены противоречивые результаты, выделяют общие признаки тех классов задач, для которых новый алгоритм работает эффективнее существующих.

Можно сформулировать общие требования к достаточно эффективному алгоритму поиска:

- при известном решении задачи этот алгоритм должен выдавать абсолютно точное решение, не используя перебора,

- при абсолютно неизвестной задаче алгоритм должен использовать полный перебор, на основе алгоритмов поиска в ширину или в глубину,

- при решении задачи, фрагменты которой являются ранее решенными, для них необходимо использовать готовое решение, в то время как для оставшихся фрагментов нужно использовать перебор,

- при существовании в решаемой задаче аналогий с другими ранее решенными, необходимо пользоваться аналогичной оценочной функцией.

Перечисленные требования исчерпывают все возможности, которые могут быть использованы для получения максимально точного и универсального алгоритма поиска решения. В то же время, формулирование этих требований явно выделяет тип поиска, который в этом случае должен рассматриваться, а именно - поиск в пространстве задач, который предполагает разбиение задач на подзадачи с последующим построением плана их объединения в единое решение. Этот тип поиска имеет различные реализации в рамках иерархических решателей задач или интеллектуальных систем ситуационного управления и квалитативной физики [2].

Контрольные вопросы к главе3

1. Какие варианты программной реализации эвристических программ вы знаете?
2. Какое понятие используется в продукционных системах вместо понятия “ход”? В чем смысл этого понятия?
3. Перечислите недостатки программы Simple_Heuristic.
4. При каких условиях заканчивается программа Simple_Heuristic?
5. В чем преимущества и недостатки программы Gradient_Heuristic?
6. Как реализуется в программе Gradient_Heuristic построение результата поиска? Из чего этот результат состоит?
7. Перечислите основные достоинства программы Intelligent_Heuristic.
8. Для чего в программе Intelligent_Heuristic используется список ситуаций, какова структура этого списка, какая ситуация располагается в нем на первом месте?
9. Что “забывает” программа Intelligent_Heuristic? Чем это плохо или хорошо?
10. Перечислите производные характеристики поиска.

Г Л А В А 4

ФОРМАЛЬНОЕ ИССЛЕДОВАНИЕ ИНТЕЛЛЕКТУАЛЬНЫХ РЕШАТЕЛЕЙ ЗАДАЧ

4.1. Исследование эвристических программ

Эвристическое программирование появилось в методологии интеллектуальных систем значительно ранее, чем теория представления знаний, экспертных, расчетно-логических и естественно-языковых систем [3]. Благодаря времени появления и эффективности методов эвристическая составляющая присутствует во всех классах интеллектуальных программ. Вместе с тем, существуют программы, построенные исключительно на методах эвристического проектирования. Такие программные системы принято называть *интеллектуальными решателями задач*.

Методология интеллектуальных решателей основывается на формулировании какой-либо задачи предметной области как описании программных моделей двух ситуаций, одна из которых является начальной, а другая - целевой. Кроме этого известно множество действий, которыми можно изменять ситуацию в предметной области. В то же время остается неизвестной последовательность выполнения действий для достижения из начальной ситуации соответствующей целевой. Поиск этих последовательностей и является предметом решения задачи эвристического проектирования.

Эвристическое проектирование является более общим понятием, чем, например, продукционное проектирование, которое является одним из его частных случаев. Современные подходы к этому классу систем ориентируются на такую архитектуру представления знаний, при которой основные характеристики поиска решения являются наиболее близкими к аналогичным характеристикам беспереборных алгоритмов. Этот факт определяет основную цель разработки *программных машин*, адекватных инструментарию эвристического программирования, как *разработку набора формальных средств для проектирования и исследования алгоритмов с наилучшими характеристиками поиска*.

Приведем далее элементы формальных программных машин, позволяющих исследовать системы, основанные на эвристическом подходе.

Алгебраическая система П для исследования изменения состояний в дереве вариантов представляет собой набор множеств $< Q, S, R >$, которые описывают возможные операции S по преобразованию ситуаций предметной области Q. На множестве состояний могут быть заданы отношения R, определяемые, например, при помощи оценочной функции, как различные меры близости текущей ситуации к целевой. В отличие от алгебр изменения состояний универсальных алгоритмических языков, для описания эвристических программ сигнатура S использует операции, задаваемые парами (q_b, q_e) ситуаций из Q . Так об операции s_i (q_b, q_e) можно сказать, что она переводит ситуацию предметной области q_b в ситуацию q_e . Очевидно, что сами по себе операции из множества S недостаточны для построения планов решений, поскольку для этого необходимо рассматривать композицию операций. В этом случае можно было бы рассмотреть другую систему множеств:

$$П = < SQ , H, R >,$$

где SQ - множество пар (q_i, q_j) $\in Q \times Q$, а H - множество операций композиции этих пар в планы решения задачи. Множество R при этом преобразуется в множество отношений над парами из SQ, которые так же, как и ранее, можно рассматривать в качестве мер предпочтения выбора той или иной пары для решения задачи с конкретной заданной целью.

В то же время, неудобство использования такой алгебраической системы в том, что при построении планов решения, операции из S должны быть определены не столько на множестве пар ситуаций, сколько на множестве самих ситуаций, что делает предложенную систему множеств менее строгой в математическом смысле.

В качестве более строгой и мощной системы множеств, направленной на исследование инструментария эвристических программ, можно предложить алгебраическую систему с метауровнем или *метаалгебраическую систему*, представляющую собой четверку множеств:

$$П = < Q , S, H, R >,$$

в которой множества Q, S и R аналогичны соответствующим множествам алгебры изменения состояний, а множество H представляет собой множество операций, определенных на множестве S. Иными словами, операции из H используются для построения планов решения конкретных задач в предметной области.

Рассмотрим множество $H = \{ h_1, ..., h_n \}$. Его можно разделить на два подмножества, в одном из которых будут присутствовать операции обобщения элементов из S и Q, а во втором - операции последовательной и альтернативной композиции элементов S.

Сложность операций из H определяется сложностью синтаксиса конкретного алгоритмического языка или языка описания ситуаций в предметной области. К языкам этого типа можно отнести языки семантического представления, язык ситуационного управления, фреймовые языки [2], универсальный семантический код и т.п. . При проектировании эвристических программ, основанных на планировании решения, необходимо использование качественно новых операций, в частности, *операций выделения наиболее общих задач* и *последовательной или альтернативной композиции общих задач*.

Предложим наиболее общую постановку задачи и способ ее решения для построения семантики операций множества H. Носитель Q является множеством описания ситуаций предметной области в некоторой программной модели. Программная модель может строиться различным образом, но всегда представляет собой систему элементов памяти, а следовательно, общий вид такого элемента можно представить следующим образом:

$$q_i = (m_1, k_1), (m_2, k_2), ..., (m_n, k_n),$$

где m - имена элементов ситуации, а k - их значения. Из свойств алгебры элементов ситуации следует, что элементы могут быть составлены при помощи операции композиции из более простых элементов. Образованная таким образом псевдоиерархия элементов ситуации позволяет каждой ситуации в предметной области сопоставить единственный сложный элемент.

Наиболее простым способом определения родовидовых отношений на множестве элементов ситуации является вычисление теоретико-множественного пересечения элементов, представленных композицией простейших элементов. Этого вполне достаточно для выделения более общих и более частных ситуаций, поскольку общая часть элементов для двух ситуаций может рассматриваться как наиболее общая ситуация для этих двух ситуаций. Если соответствующее родовидовое отношение обозначить через $IsA (q_i, q_j)$, читающееся как "ситуация q_j является частным случаем ситуации q_i ", то можно получить условие построения родовидовых отношений на множестве операций из S. Оно выглядит следующим образом:

$$IsA (q_a, q_d) \& IsA(q_c, q_f) \exists s_x \in S, s_x (q_a, q_c) \Rightarrow$$
$$\exists s_y \in S, s_y (q_d, q_f) \& IsA (s_y, s_x) .$$

Это выражение задает наиболее общий случай установления родовидового отношения между операциями, в котором не учитываются ранги ситуаций как случаи опосредованных родовидовых отношений. Для упрощения, выражение содержит одинаковое обозначение отношения между ситуациями предметной области и операциями по их преобразованию.

Более точное условие определения отношения род-вид может учитывать ранг отношения:

$$IsA (q_a, q_d) \& IsA(q_c, q_f) \& \neg \exists q_r, q_t , IsA (q_a, q_r) , IsA(q_r, q_d),$$
$$IsA(q_c, q_t), IsA(q_t, q_f) \& \exists s_x \in S, s_x (q_a, q_c) \Rightarrow$$
$$\exists s_y \in S, s_y (q_d, q_f) \& IsA (s_y, s_x) .$$

Заметим, что преимущество построения таких родовидовых отношений для решения задач эвристического программирования в том, что для каждой из операций преобразования ситуаций более высокого ранга может существовать несколько более частных операций, т.е.

$$\exists n\, IsA (q_a, q_d), \exists n\, IsA(q_c, q_f) [\neg \exists q_r, q_t , IsA (q_a, q_r) , IsA(q_r, q_d),$$
$$IsA(q_c, q_t), IsA(q_t, q_f) \& \exists s_x \in S, s_x (q_a, q_c)] \Rightarrow$$
$$\exists s_y \in S, s_y (q_d, q_f) \& IsA (s_y, s_x) ,$$

где символы $\exists n$ читаются как "существует ровно n таких элементов (отношений)".

Сделанные определения позволяют конструктивно формировать планы решения эвристических задач на основе алгоритма сопоставления действий по преобразованию ситуаций предметной области задачи. Приведем этот алгоритм в общем виде.

I) Построить псевдоиерархию возможных ситуаций предметной области до некоторого заданного нижнего уровня n, для чего выполнить следующие действия.

1) Сформировать все пары ситуаций, связанных отношением IsA, т.е. имеющих общие составляющие согласно ранее данным определениям.

2) На основе общих вершин пар построить псевдоиерархический граф.

II) Построить псевдоиерархию последовательностей конкретных действий преобразования ситуаций в предметной области (аналогичной дереву вариантов), для чего выполнить следующее.

1) Объединить вершины, соответствующие равными начальными ситуациями в поддеревья с общим корнем.

2) Произвести композицию подграфов с учетом совпадения начальных и заключительных вершин действий полученных подграфов.

III) Построить план решения задач, произведя композицию двух построенных ранее графов, для чего выполнить следующие действия.

1) Сформировать все пары действий, начальная и заключительная ситуации которых находятся в непосредственном IsA отношении, т.е. имеют общие составные элементы согласно заданным выше определениям.

2) Действия, начальные состояния которых эквивалентны, представить набором ребер графа псевдоиерархии с общей начальной вершиной.

3) Произвести последовательную композицию полученных на шаге 2 подграфов с учетом совпадения корневых вершин для IsA-отношений более низких рангов с заключительными вершинами IsA-отношений более высоких рангов.

Следует заметить, что определения и алгоритм, приведенные в настоящем параграфе могут быть более эффективными, если на проектировщика эвристической программы возложить функции интуитивного сопоставления ситуаций и шагов решения с выделением общих частей. С точки зрения формальной программной машины в этом случае в множество действий S должны быть добавлены обобщающие действия более сложной структуры, чем теоретико-множественное пересечение элементов. Для автоматического выделения таких обобщений можно включить в систему элементов памяти, определяющих ситуации, интуитивно относимые программистом к одному классу, новые равные элементы, соответствующие именам класса. Это гарантирует отыскание общих элементов для всех действий одного и того же класса.

Таким образом, на основе формализма программных машин показан общий способ построения оптимизированных эвристических программ, превосходящих реализации, рассмотренные в предыдущих главах, на основе использования иерархических (псевдоиерархических) планов решений.

4.2. Строгое описание понятия “продукция”

Формализм продукции принадлежит Посту [4] и весьма жестко связан с формализмом машин Тьюринга. Пост доказал функциональную эквивалентность продукционной и тьюринговской машин.

Пусть существует некоторое слово (цепочка) a. Пусть необходимо найти первое вхождение в него другого слова b и заменить его третьим словом c. Эту функцию может выполнить операция подстановки, обозначаемая Sb(a, b, c).

Пусть b,c - некоторые слова в алфавите, не содержащем символа “$\rightarrow$”. Предложением $b \rightarrow c$ обозначают команду подстановки следующего содержания: “в заданное входное слово вместо первого вхождения слова b подставить слово c”. Выражения этого типа принято называть *командами подстановки* или *подстановками*.

Применение подстановки $b \rightarrow c$ к некоторому слову a, содержащему слово b, будет иметь результатом новое слово Sb(a, b, c). Если слово b не входит в слово a, то результат подстановки считается неопределенным.

Состояние машины Тьюринга может записываться одним словом, записанным в алфавите, являющемся объединении внешнего (предназначенного для записи в ячейки ленты машины Тьюринга) и внутреннего (обозначений состояний машины) алфавитов. Это слово называется *машинным словом* и имеет следующий вид: $x_1\ x_2\ \ldots\ q_k\ x_i\ \ldots\ x_n$, где x_i - какие-либо значения из внешнего алфавита машины $a_0, a_1, \ldots, a_n$, характеризующая текущие состояния всех ячеек ленты в определенный момент времени, а q_k - текущее состояние внутренней памяти. Символ q_k входит в машинное слово всего один раз и указывает своим местоположением текущую читаемую ячейку ленты. Символ-состояние этой ячейки располагается сразу же *за* символом текущего состояния внутренней памяти.

Определим границы машинного слова, чтобы знать, когда необходимо достраивать ленту машины Тьюринга новыми ячейками. Для этого введем специальные символы начала и конца машинного слова, соответственно u, v.

В этом случае *расширенное машинное слово* m , будет записываться как umv.

Предположим, что машина Тьюринга содержит команду $q_x\ a_i \rightarrow q_y\ a_j$, читаемую как "находясь в состоянии q_x и напротив ячейки ленты машины Тьюринга с содержимым a_i, перейти в состояние q_y записав в ячейку содержимое a_j". Пусть расширенное машинное слово содержит подслово $q_x\ a_i$, и пусть есть расширенное машинное слово um^1v, содержащее подслово $q_y\ a_j$.

Тогда можно сказать, что слово um^1v получается из слова umv путем выполнения команды подстановки $q_x\ a_i \rightarrow q_y\ a_j$.

Аналогичным образом несложно через подстановки определить команды левого и правого сдвига (приписывания ячейки) ленты: $q_x\ a_i \rightarrow q_y\ L$, $q_x\ a_i \rightarrow q_y\ R$.

Учитывая изложенное, программа машины Тьюринга может представлять собой набор команд подстановок.

Для этого случая Постом было сформулировано следующее правило. Для того, чтобы из расширенного машинного слова umv получить новое слово um^1v, описывающее непосредственно следующее состояние машины Тьюринга, достаточно над словом umv выполнить ту из команд подстановки, которая применима к входному слову. Такая команда в программе может быть только одна. Если же ни одна из команд не применима к слову umv, то m содержит символ заключительного состояния внутренней памяти q_0, и следовательно слово umv описывает заключительное состояние машины Тьюринга.

Системы продукций были существенно развиты в интеллектуальных системах. При этом продукцией может считаться некоторая четверка вида N: Q, S $\rightarrow$ P , где N - имя продукции Q - *область применимости* продукции, S - ее *антецедент*, а P - *консеквент*. Имя является уникальным идентификатором.

Область применимости означает обычно некоторый слабый предикат, характеризующий предметную область задачи, для фрагмента решения которой предназначена продукция. Антецедент - сильный предикат (логическое выражение), принимающее истинность на множестве элементов памяти продукционной программы. Истинность антецедента указывает на необходимость выполнения действия S по преобразованию элементов памяти. Такая простая форма продукции, тем не менее может предполагать весьма сложный синтаксис и семантику для составляющих Q, S и P, которые могут включать в себя сложные алгоритмы в качестве составляющих.

Некоторые элементы продукций могут иметь характер метапреобразований, например, удаления какой-либо продукции или изменение приоритета при выборе. Вместе с тем, какой бы сложной не была структура N: Q, S $\rightarrow$ P, ее всегда можно подвергнуть декомпозиции, получив несколько более простых продукций.

4.3. Исследование продукционных систем

Эвристическая сложность программных систем, построенных на основе продукций, заключается в том, что в базовом продукционном формализме на каждом из шагов решения может быть использовано любое правило из всех имеющихся. Это приводит к большому числу переборов вариантов для определения условия применимости правила. Кроме того, допускаются ситуации, когда могут быть применимы сразу же несколько правил. В этом случае приходится руководствоваться различными *стратегиями* выбора продукций, в том числе *стратегиям разрешения* правил [5].

Рассмотрим применение формализма программных машин для продукционного инструментария. Для алгебры элементов памяти таких машин используют, как правило, какую-либо из *алгебр фреймов*.

Фреймы можно представлять как сложные элементы памяти, состоящие из строковый ячеек - *слотов*. Для этих элементов существуют отношения наследования и "часть-целое", как и для инструментария при объектно-ориентированном проектировании.

Основными формализмами продукционных системах являются алгебры продукций. Этот тип позволяет довольно просто достраивать и объединять различные продукционные программы, используя операцию объединения. В то же время для сокращения *пространства состояний* эвристического перебора продукций используют также мультипликативную операцию, которая в этом случае является операцией последовательной композиции продукций.

Действительно, рассмотрим некоторое ограниченное множество продукционных правил P = { p_1, p_2, ..., p_n }. Это множество является множеством-носителем алгоритмической алгебры - решетки. Если предположить, что в алгебре используется лишь одна аддитивная операция объединения множества продукций, то алгебра продукций образует полугруппу.

Пусть также есть еще одно множество продукций Q = {q_1, q_2, ..., q_m }, также образующее полугруппу. Будем рассматривать вариант, при котором необходимо выбирать на каждом шаге решения все продукции, которые могут быть применены.

Если алгоритм P решает какую-либо задачу за k шагов, то общее число продукций, которое ему необходимо будет просмотреть на применимость, будет равным k*n, поскольку на каждом из шагов приходится рассматривать n продукций. Для программы Q это число также для k шагов решения соответственно составит величину k*m. Если необходимо объединить программы P и Q, приходится использовать операцию объединения множеств правил: P + Q = { p_1, p_2, ..., p_n, q_1, q_2, ..., q_m }.

При отсутствии в объединяемых множествах эквивалентных продукций общее число элементов результирующего множества станет равным n+m. Теперь для решения той же самой задачи в k шагов результирующая программа выполнит (n + m) * k сопоставлений. Если предположить, что никакое из правил множества Q не может выполняться ранее, чем правила из P, то приходится признать неэффективность аддитивной операции композиции, поскольку выполнение последовательной мультипликативной композиции оказывается более целенаправленным.

Последовательная композиция P*Q предполагает, что после выполнения шагов правилами множества P, они больше не рассматриваются, а сопоставление происходит лишь для правил множества Q. Из этого

следует, что общее число просмотров антецедентов для k шагов решения станет равным x*n + y*m = k, где x + y = k .

Заметим, что некоторые продукционные системы используют удаление из множества рассматриваемых правил каждого отработавшего правила. Несложно увидеть, что в этом случае число просмотров антецедентов для параллельной и последовательной композиций хоть и имеет меньшую разницу, чем в предыдущем случае, но эта разница остается все равно в пользу случая мультипликативной композиции. Сказанное свидетельствует о целесообразности выделения в системах продукций последовательных композиций.

Рассмотренное сравнение операций композиции делает возможным анализ продукционных программ с целью их оптимизации. Пусть есть некоторая продукционная программа S, представленная множеством продукций $\{ s_1, ..., s_n \}$. В этом множестве можно выделить правила, которые должны следовать друг за другом, т.е. не может быть ситуации, чтобы левая часть одного их правил получила бы значение истинности перед тем, как сработало второе правило. В этом случае эти два правила можно отнести к функционально зависимым и считать, что между ними существует соответствующее отношение, и может быть применена операция мультипликативной композиции. Для каждой из продукций можно выделить множества продукций, которые разделяются на следующие классы:

- множество обязательно предшествующих продукций,
- множество продукций, которым обязательно предшествует рассматриваемая,
- правила, которые никак не влияют на выполнении продукции,
- правила, влияющие на выполнение продукции, но не предшествующие и последующие относительно продукции в обязательном порядке (т.е. правила, которые могут быть выполнены и до и после рассматриваемой продукции).

Такая классификация продукций по их функциональной зависимости позволяет выделить *конъюнктивную нормальную форму* (КНФ) продукционной программы. КНФ представляет собой последовательную композицию множеств продукционных правил, для которых выполнение продукций внутри этих множеств не носит выраженного последовательного характера.

Перечислим конструктивные признаки для классификации продукционные правил:

- если два правила используют одни и те же элементы памяти (фреймы и слоты), то они являются функционально зависимыми,
- если для двух правил выполнение консеквента одного из них влечет выполнение антецедента другого, то эти правила находятся в отношении обязательного следования,
- если продукции не содержат ни одного общего элемента памяти в области определения и области значений, то они непосредственно функционально независимы,
- если две продукции непосредственно функционально независимы и в программе не существует последовательности непосредственно зависимых продукций, связывающих эти две продукции, то они являются полностью функционально независимыми.

Перечисленные признаки имеют свои следствия, позволяющие оптимизировать программы. Так, например, полностью функционально независимые правила можно относить к правилам, предназначенным для решения задач из разных предметных областей или задач различной направленности. Такое разделение требует единственного общего слабого условия-антецедента для целого множества конструкций. Это условие можно выделить в одну единственную продукцию-предшественник, выполнение условия левой части которой приведет к последующему перебору при решении задачи лишь тех правил, которые относятся к соответствующей предметной области.

Последовательное применение оптимизирующих преобразований к системам продукций приводит к образованию псевдоиерархий для всех левых и правых частей продукций. При этом число просмотров левых частей продукций стремится к длине пути решения. В этом случае становится явно выраженной эвристическая составляющая, которая заключается в организации перебора вариантов на псевдоиерархических структурах в соответствии с выбранной стратегией перебора.

Контрольные вопросы к главе 4

1. Что такое “программная машина”? Из каких элементов она состоит?
2. Из чего состоит алгебраическая система П для исследования изменения состояний в дереве вариантов?
3. Какие операции используются в алгебрах для исследования эвристических систем?
4. За счет чего при алгебраическом исследовании эвристических и продукционных систем повышается их эффективность?