Московский государственный университет им. М.В. Ломоносова
Факультет вычислительной математики и кибернетики

Е.И. Большакова, М.Г. Мальковский, В.Н. Пильщиков

ИСКУССТВЕННЫЙ ИНТЕЛЛЕКТ. АЛГОРИТМЫ ЭВРИСТИЧЕСКОГО ПОИСКА

(Учебное пособие)

Москва
2002

УДК 519.68+681.142
ББК 22.18

Большакова Е. И., Мальковский М. Г., Пильщиков В. Н. Искусственный интеллект. Алгоритмы эвристического поиска (учебное пособие) — М.: Издательский отдел факультета ВМК МГУ (лицензия ИД № 05899 от 24.09.01), 2002.–83 с.

В пособии излагаются основные понятия и алгоритмы теории эвристического поиска, представляющей одно из классических направлений исследований в области искусственного интеллекта. Пособие предназначено для студентов факультета ВМК МГУ в поддержку основного курса «Искусственный интеллект», а также для студентов и аспирантов программистских специальностей.

Авторы благодарят Н.Э. Васильеву за помощь в подготовке пособия.

Рецензенты: доцент В.Г. Баула
доцент Л.С. Корухова

Печатается по решению Редакционно-издательского Совета факультета вычислительной математики и кибернетики МГУ им. М.В. Ломоносова.

ISBN 5-89407-150-X

ОГЛАВЛЕНИЕ

АННОТАЦИЯ

Данное пособие представляет собой введение в теорию эвристического поиска — один из наиболее разработанных разделов области исследований, известной под названием «искусственный интеллект».

Как и в других учебниках, в данном пособии основные понятия и методы этой теории иллюстрируются примерами известных игр, головоломок и математических задач. У этой традиции, сложившейся почти за 30 лет существования теории эвристического поиска, существует важная причина. Как пишет один из создателей теории Н. Нильсон, «Игры и математические задачи берутся не потому, что они просты и ясны, а потому, что они при минимальных структурах дают нам наибольшую сложность... В игровых задачах и решениях головоломок возникли и отшлифовались многие идеи, которые оказались по-настоящему полезными для менее легкомысленных задач» [Нильсон73, с.13].

Двумя составными элементами процесса решения задач в теории эвристического поиска является *представление* (формализация) задач и собственно решение — *поиск*. В пособии рассматриваются два подхода к решению задач и, соответственно, два способа представления — подход с использованием пространства состояний и подход, основанный на редукции задач. Для обоих подходов описываются используемые алгоритмы поиска решения. Важной особенностью большинства этих алгоритмов является использование *эвристической* информации. *Эвристикой* обычно принято называть любое правило, стратегию, прием, существенно помогающий решению некоторой задачи. В области искусственного интеллекта и теории поиска под эвристической информацией понимается все то, что относится к конкретной решаемой задаче и служит более эффективному ее решению.

ГЛАВА 1. ПОДХОДЫ К РЕШЕНИЮ ЗАДАЧ

1.1. Представление задач в пространстве состояний

1.1.1. Основные понятия

Типичным представителем класса задач, для которых подходит представление в пространстве состояний, является головоломка, известная как игра в пятнадцать — см. рис.1(а). В ней используется пятнадцать пронумерованных (от 1 до 15) подвижных фишек, расположенных в клетках квадрата 4×4. Одна клетка этого квадрата всегда остается пустой, так что одну из соседних с ней фишек можно передвинуть на место этой пустой клетки, изменив тем самым местоположение пустой клетки. Заметим, что более простым вариантом этой головоломки является игра в восемь (квадрат 3×3 и восемь фишек) – пример соответствующей задачи показан на рис.1(б).

На рис.1(а) изображены две конфигурации фишек. В головоломке требуется преобразовать первую, или начальную, конфигурацию во вторую, или целевую конфигурацию. Решением этой задачи будет подходящая последовательность сдвигов фишек, например: передвинуть фишку 8 вверх, фишку 6 влево и т.д.

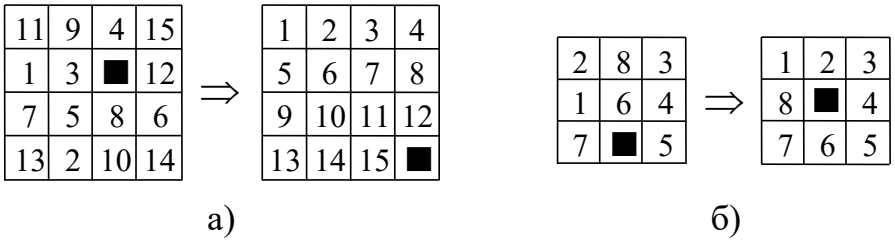

Рис. 1. Игра в пятнадцать и игра в восемь фишек

Важной особенностью класса задач, к которому принадлежит игра в пятнадцать (восемь) фишек, является наличие в задаче точно определенной начальной ситуации и точно определенной цели. Имеется также некоторое множество операций, или ходов, переводящих одну конфигурацию в другую. Именно из таких ходов состоит искомое решение задачи, которое можно в принципе получить методом проб и ошибок. Действительно, отправляясь от начальной ситуации, можно построить конфигурации, возникающие в результате выполнения возможных в этой ситуации ходов, затем построить множество конфигураций, получающихся после применения следующего хода, и так далее – пока не будет достигнута целевая конфигурация.

Введем теперь основные понятия, используемые при формализации задачи в пространстве состояний. Центральным из них является понятие ***состояния*** задачи. Например, для игры в пятнадцать (или в восемь) состояние — это просто некоторая конкретная конфигурация фишек.

Среди всех состояний задачи выделяются начальное состояние и целевое состояние, в совокупности определяющие задачу, которую надо решить — примеры их приведены на рис. 1.

Другим важным понятием является понятие оператора, т.е. допустимого хода в задаче. Оператор преобразует одно состояние в другое, являясь по сути функцией, определенной на множестве состояний и принимающей значения из этого множества. Для игры в пятнадцать или в восемь удобнее выделить четыре оператора, соответствующие перемещениям пустой клетки (фишки «пустышки») влево, вправо, вверх, вниз. В некоторых случаях оператор может оказаться неприменимым к какому-то состоянию: например, операторы сдвига вправо и вниз неприменимы, если пустая клетка расположена в правом нижнем углу. Значит, в общем случае оператор является частично определенной функцией отображения состояний.

В терминах состояний и операторов решение задачи есть определенная последовательность операторов, преобразующая начальное состояние в целевое. Решение задачи ищется в пространстве состояний — множестве всех состояний, достижимых из начального состояния при помощи заданных операторов. Например, в игре в пятнадцать или в восемь пространство состояний состоит из всех конфигураций фишек, которые могут быть образованы в результате возможных перемещений фишек.

Пространство состояний можно представить в виде направленного графа, вершины которого соответствуют состояниям, а дуги (ребра) — применяемым операторам. Указанные в виде стрелок направления соответствуют движению от вершины-аргумента применяемого оператора к результирующей вершине. Тогда решением задачи будет путь в этом графе, ведущий от начального состояния к целевому. На рис.2 показана часть пространства состояний для игры в пятнадцать: в каждой вершине помещена та конфигурация фишек, которую она представляет. Все дуги между вершинами являются двунаправленными, поскольку в этой головоломке для любого оператора есть обратный ему (точнее, множество операторов состоит из двух пар взаимно-обратных операторов: влево-вправо, вверх-вниз).

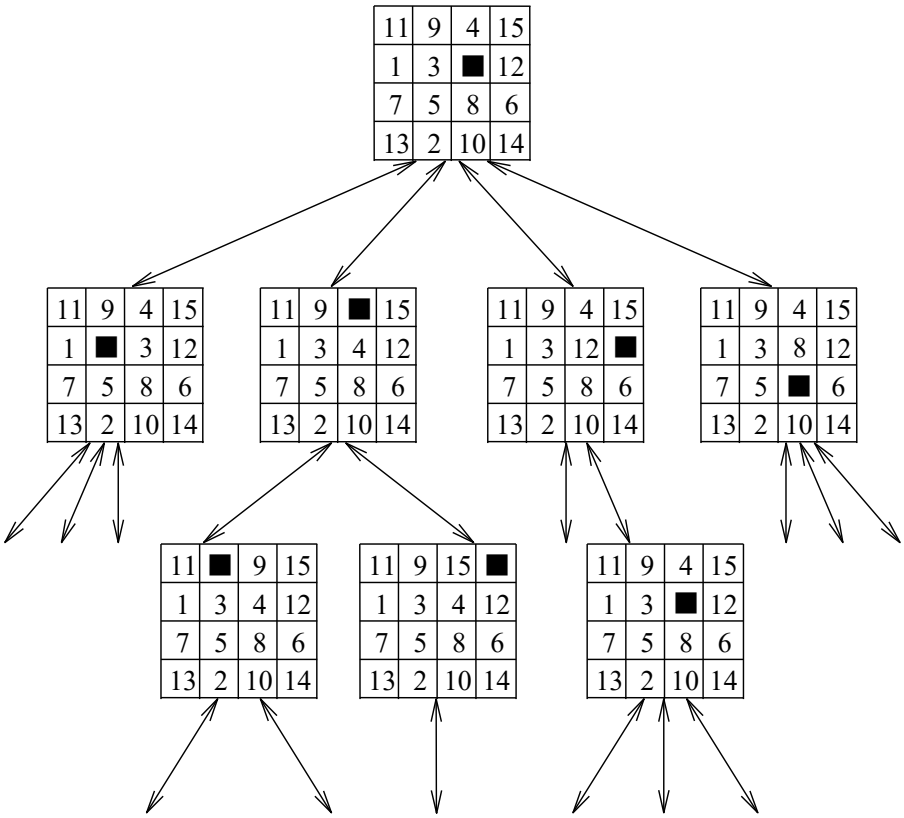

Рис. 2. Часть пространства состояний для игры в пятнадцать

Пространства состояний могут быть большими и даже бесконечными, но в любом случае предполагается счетность множества состояний.

Таким образом, в подходе к решению задачи с использованием пространства состояний задача рассматривается как тройка (S_I , O, S_G) , где:

- S_I — начальное состояние;
- O — конечное множество операторов, действующих на не более чем счетном множестве состояний;
- S_G — целевое состояние.

Дальнейшая формализация решения задачи с использованием пространства состояний предполагает выбор некоторой конкретной формы описания состояний задачи. Для этого могут применяться любые подходящие структуры — строки, массивы, списки, деревья и т.п. Например, для игры в пятнадцать или восемь наиболее естественной формой описания состояния будет двумерный массив. Заметим, что от выбора формы описания состояния зависит в общем случае сложность задания операторов задачи, которые должны быть также определены при формализации задачи в пространстве состояний.

Если для игры в пятнадцать средством формализации выступает язык программирования Лисп или Паскаль, то операторы задачи могут быть описаны в виде четырех соответствующих функций языка. При использовании же продукционного языка, эти операторы задаются в виде правил продукций вида: «исходное состояние → результирующее состояние» (см. [Нильсон73, раздел 2.2]).

В рассмотренных на рис.1 примерах задач искомое целевое состояние задается явно, т.е. известно местоположение каждой фишки в целевой конфигурации. В более сложных случаях игры может быть несколько целевых состояний, либо же целевое состояние может быть определено неявно, т.е. охарактеризовано некоторым свойством, например, как состояние, в котором сумма номеров фишек в верхнем ряду не превосходит 10. В подобных случаях свойство, которому должно удовлетворять целевое состояние, должно быть описано исчерпывающим образом, к примеру, путем задания булевской функции, реализующей проверку нужного свойства состояния задачи.

Итак, для представления задачи в пространстве состояний необходимо определить следующее:

- форму описания состояний задачи и описание начального состояния;
- множество операторов и их воздействий на описания состояний;
- множество целевых состояний или же описание их свойств.

Перечисленные составляющие задают неявно граф-пространство состояний, в котором необходимо найти решение задачи. Заметим попутно, что, в отличие от такого неявного способа задания графа, при явном способе задания все вершины и дуги графа должны быть перечислены, например, с помощью таблиц.

Решение задачи в пространстве состояний подразумевает просмотр неявно заданного графа, для чего необходимо преобразование в явную форму достаточно большой его части, включающей искомую целевую вершину. Действительно, просмотр осуществляется как последовательный ***поиск***, или ***перебор*** вершин, в ***пространстве состояний***. В исходной точке процесса к начальному состоянию применяется тот или иной оператор и строится новая вершина-состояние, а также дуги, связывающие ее с корневой вершиной. На каждом последующем шаге поиска к одной из уже полученных вершин-состояний применяется допустимый оператор и строится еще одна вершина графа и связывающие дуги. Если очередная построенная вершина соответствует целевому состоянию, то процесс поиска завершается.

1.1.2 Примеры пространств состояний

Разберем два характерных примера представления задач в пространстве состояний, показывающих, что такое представление возможно для различных типов задач. Подчеркнем заранее, что предлагаемые ниже представления, хотя и являются достаточно естественными, все же не являются единственно допустимыми в этих задачах, возможны и другие варианты.

Вообще, от выбора представления, т.е. рассмотренных выше составляющих, зависит размер пространства состояний, а значит, и эффективность поиска в нем. Очевидно, желательны представления с малыми пространствами состояний, но нахождение удачных представлений, сужающих пространство поиска, требует обычно некоторого дополнительного анализа решаемой задачи.

Рассмотрим формализацию в пространстве состояний известной задачи о коммивояжере. Коммивояжер, располагая картой дорог, соединяющей 7 городов (см. рис.3), должен построить свой маршрут так, чтобы, выехав из города *A*, он посетил каждый из других шести городов *B*, *C*, *D*, *E*, *H*, *G* в точности по одному разу и затем вернулся в исходный город. В другом, более сложном варианте задачи требуется также, чтобы маршрут имел минимальную протяженность.

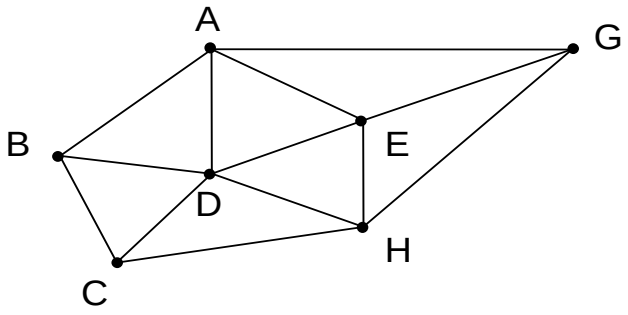

Рис. 3. Задача о коммивояжере: схема дорог между городами

Состояние решаемой задачи можно задать как список городов, которые коммивояжер уже посетил к текущему моменту. Тогда возможным состояниям соответствуют списки из элементов *A*, *B*, *C*, *D*, *E*, *H*, *G* без повторений, исключение составляет только элемент-город *A*, он может встретиться в списке дважды – в начале списка и его конце. Пример списка-состояния — (*A B C H*). Начальное же состояние определяется как список (*A*), а целевое — как любой допустимый список, начинающийся и кончающийся элементом *A*.

Для определенных таким образом состояний операторы задачи могут соответствовать перемещениям между городами (т.е. дугам графа рис.3) — получаем таким образом 13 операторов. На рис.4 показана в виде графа часть получающегося пространства состояний, включающая решающий путь. Этот граф является деревом – действительно, при применении любого оператора список уже посещенных городов может только удлиниться, поэтому в процессе построения новых вершин не может возникнуть вершина, встречавшаяся прежде.

Операторы в этой задаче могут быть определены и другим образом. Например, введем для каждого из семи городов оператор переезда в этот город (из любого другого). Такой оператор применим к некоторому описанию состояния, если он соответствует карте дорог и в список-описание не включен город, в который надо произвести переезд. Например, оператор переезда в город *B* неприменим к состоянию (*A B C D*), но применим к состоянию (*A D*).

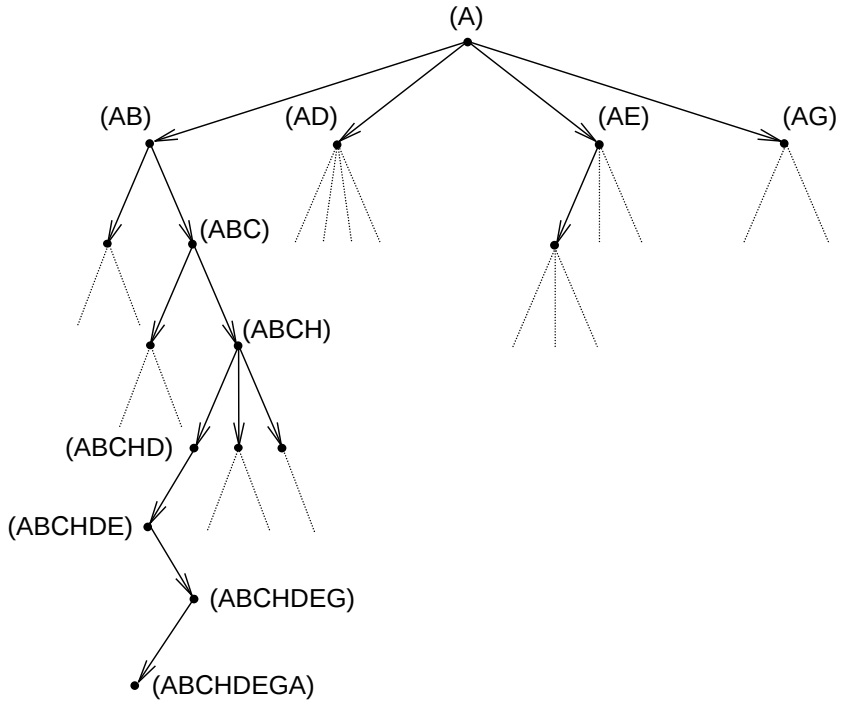

Рис. 4. Задача о коммивояжере: часть пространства состояний

Обратимся теперь к известной задаче об обезьяне и банане, простейшую формулировку которой мы и рассмотрим. В комнате находятся обезьяна, ящик и связка бананов, которая подвешена к потолку настолько высоко, что обезьяна может до нее дотянуться, только встав на ящик. Нужно найти последовательность действий, которые позволят обезьяне достать бананы. Предполагается, что обезьяна может ходить по комнате, двигать по полу ящик, взбираться на него и хватать бананы.

Ясно, что описание состояния этой задачи должно включать следующие сведения: местоположение обезьяны в комнате — в горизонтальной плоскости пола и по вертикали (т.е. на полу она или на ящике), местоположение ящика на полу и наличие у обезьяны бананов. Все это можно представить в виде 4-элементного списка (*ПолОб*, *ВертОб*, *ПолЯщ*, *Цель*), где

- *ПолОб* — положение обезьяны на полу (это может быть двухэлементный вектор координат);
- *ПолЯщ* — положение ящика на полу;
- *ВертОб* — это символ *П* или *Я* в зависимости от того, где находится обезьяна, на полу или на ящике;

Цель — это число 0 или 1 в зависимости от того, достала ли обезьяна бананы или нет.

Зафиксируем также три следующие значимые точки в плоскости пола:

- $T_О$ – точка первоначального местоположения обезьяны;
- $T_Я$ – точка первоначального расположения ящика;
- $T_Б$ – точка пола, расположенная непосредственно под связкой бананов.

Тогда начальное состояние задачи описывается списком ($T_О$, $П$, $T_Я$, 0), а целевое состояние задается как любой список, последний элемент которого равен 1.

Операторы в этой задаче можно определить так:

– *Перейти* (W) — обезьяна переходит в точку W плоскости пола;
– *Передвинуть* (V) — обезьяна передвигает ящик в точку V пола;
– *Взобраться* — обезьяна взбирается на ящик;
– *Схватить* — обезьяна хватает связку бананов.

Условия применимости и действие этих четырех операторов легко задать в виде правил продукций вида:

аргумент оператора → *результат оператора,*

причем X, Y, Z, W, V — это переменные:

1. *Перейти* (W) : $(X, П, Y, Z) \rightarrow (W, П, Y, Z)$
2. *Передвинуть* (V) : $(X, П, X, Z) \rightarrow (V, П, V, Z)$
3. *Взобраться* : $(X, П, X, Z) \rightarrow (X, Я, X, Z)$
4. *Схватить* : $(T_Б, Я, T_Б, 0) \rightarrow (T_Б, Я, T_Б, 1)$

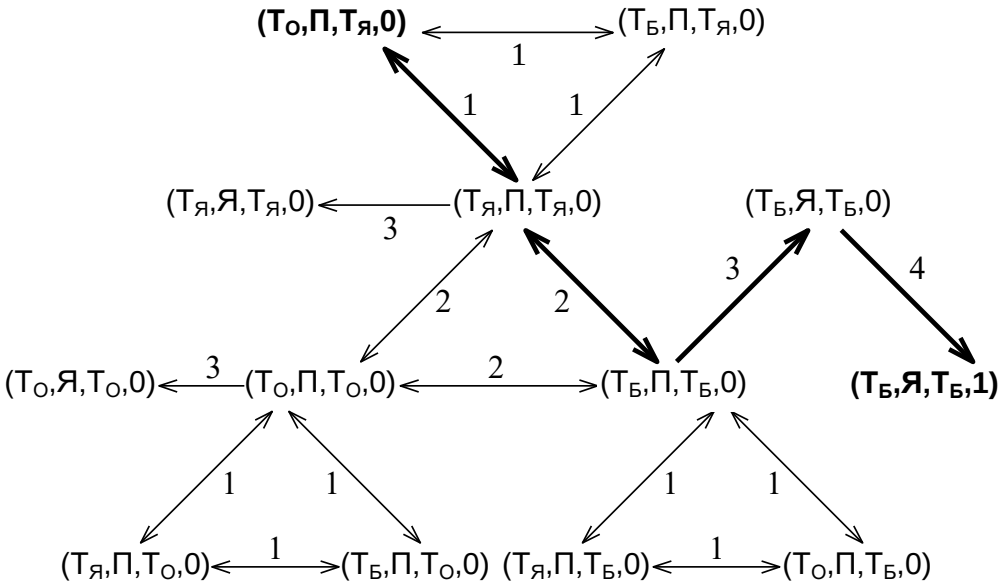

Рис. 5. Пространство состояний в задаче об обезьяне

Если считать, что для решения задачи значимы лишь вышеупомянутые точки пола $T_О$, $T_Я$, $T_Б$, тогда получим пространство состояний задачи, изображенное на рис.5. Это пространство содержит только 13 состояний, дуги графа-пространства помечены порядковым номером применяемого оператора. Пространство содержит четыре цикла хождения обезьяны между тремя значимыми точками (с ящиком или без него). В пространстве есть также две тупиковые ветви — когда обезьяна залезает на ящик, но не под связкой бананов. Жирными дугами (стрелками) показан решающий путь, состоящий из четырех операторов:

Перейти ($T_Я$); Передвинуть ($T_Б$); Взобраться; Схватить.

Рассмотренный пример показывает, сколь важен для успешного и эффективного решения задачи выбор определенного представления. Такое небольшое по размерам пространство состояний получено, в частности, вследствие того, что игнорировались все точки пола, кроме трех, соответствующих первоначальному расположению обезьяны, ящика и бананов.

Мощным приемом сужения пространств состояний является применение так называемых *схем состояний* и *схем операторов*, в которых для описаний состояний и операторов используются переменные. Тем самым схема состояния описывает целое множество состояний, а не только одно, а схема оператора определяет все множество действий некоторого типа. В рассмотренном нами представлении задачи об обезьяне использовались схемы операторов, но не схемы состояний. Другое представление этой задачи с использованием как схем состояний, так и схем операторов приведено в [Нильсон73, раздел 2.7].

1.2 Редукция задач

1.2.1 Основные понятия

Кроме уже рассмотренного подхода — представления задач в пространстве состояний – для решения ряда задач возможен и другой, более сложный подход. При этом подходе производится анализ исходной задачи с целью выделения такого набора подзадач, решив которые, мы решим исходную задачу. Каждая из выделенных подзадач в общем случае является более простой, чем исходная, и может быть решена каким-либо методом, в том числе – с использованием пространства состояний. Но можно продолжить процесс, последовательно выделяя из возникающих задач свои подзадачи – до тех пор, пока не получим ***элементарные задачи***, решение которых уже известно. Такой путь называется подходом, основанным на ***сведении задач к подзадачам***, или на ***редукции задач.***

Для иллюстрации этого подхода рассмотрим один из вариантов известной головоломки — задачи о ханойской башне, или пирамидке. В ней используются 3 колышка (обозначим их буквами *A*, *B*, *C*) и 3 диска разного диаметра, которые можно нанизывать на колышки через отверстия в центре. В начале все диски расположены на колышке *A*, причем диски меньшего диаметра лежат на дисках большего диаметра — см. рис. 6 (диски перенумерованы в порядке возрастания диаметра). Требуется переместить все диски на колышек *C*, соблюдая следующие правила. Перемещать можно только самый верхний диск, и нельзя никакой диск класть на диск меньшего размера. На рис.6 показаны начальное и целевое состояния задачи о ханойской башне.

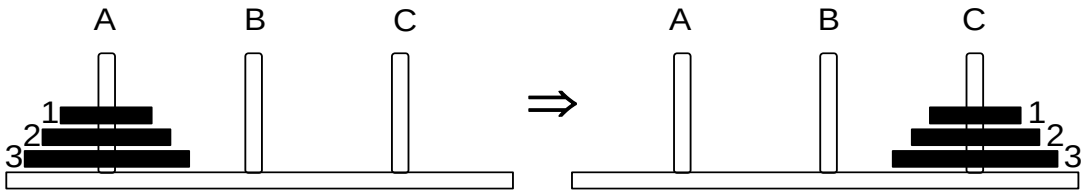

Рис. 6. Задача о ханойской башне

Эта задача легко может быть формализована в пространстве состояний: состояние задачи задается списком из трех элементов, каждый из которых указывает местоположение соответствующего диска (первый элемент — первого диска, второй – второго, третий – третьего). Начальное состояние описывается списком (*AAA*), а целевое – (*CCC*). При этом предполагается, что если на одном колышке находится более одного диска, то любой больший диск расположен ниже любого меньшего.

Полное пространство состояний задачи о пирамидке представлено графом на рис.7, включающем 27 вершин. Путь в этом графе, выделенный жирными дугами со стрелками, представляет собой решение задачи, которое включает 7 перемещений дисков.

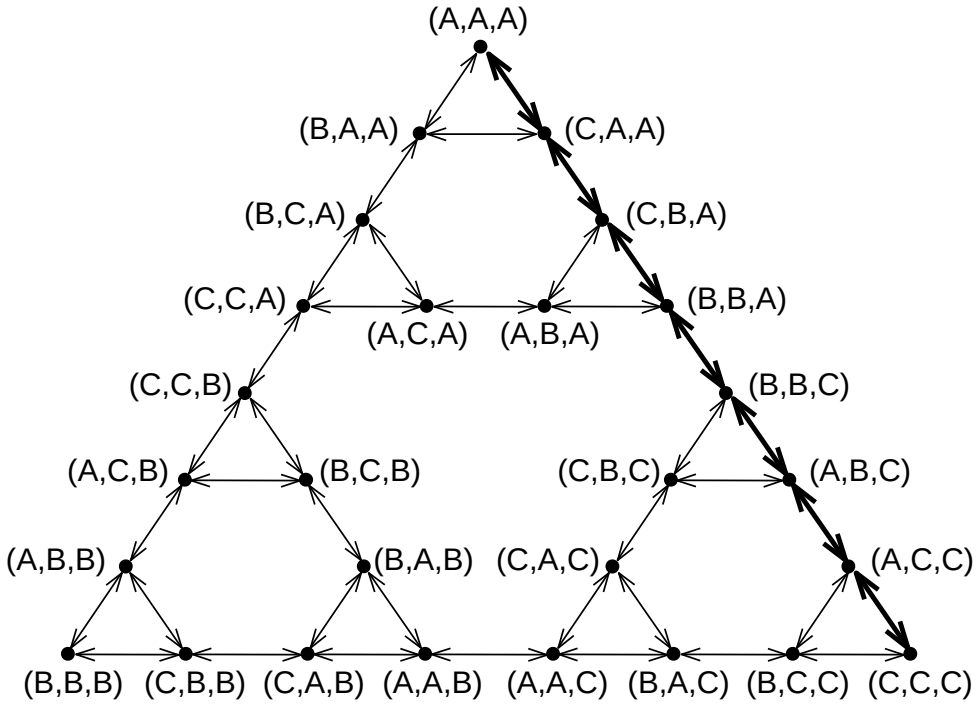

Рис. 7. Пространство состояний в задаче о ханойской башне

Это решение можно обнаружить перебором всех возможных перемещений дисков, но метод редукции задач позволяет решить рассматриваемую задачу быстрее. Ключевая идея редукции состоит в том, что для перемещения всей пирамидки необходимо переложить самый нижний диск 3, а это возможно, только если располагающаяся над ним пирамидка из двух меньших дисков 1 и 2 перенесена на колышек *B* — см. рис. 8(а).

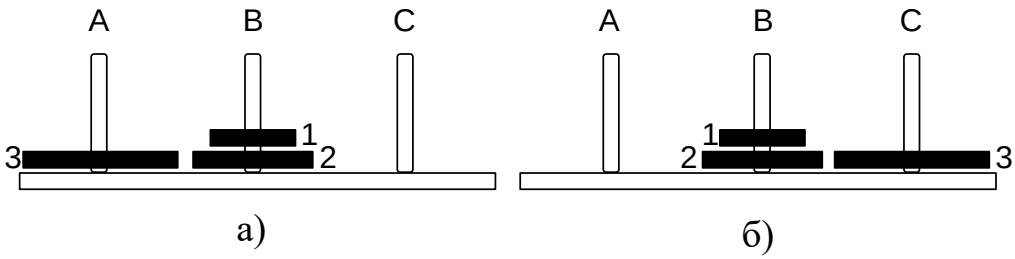

Рис. 8. Задача о ханойской башне: два решающих состояния

Таким образом, исходную задачу можно свести к трем следующим подзадачам:

1. Переместить диски 1 и 2 с колышка *A* на колышек *B* (1, 2 : *A*→*B*).
2. Переместить диск 3 с колышка *A* на колышек *C* (3 : *A*→*C*).
3. Переместить диски 1 и 2 с колышка *B* на колышек *C* (1, 2 : *B*→*C*).

На рис. 8(б) показано исходное состояние для третьей задачи.

Каждая из трех указанных задач проще исходной. Действительно, в первой и в третьей задачах требуется переместить всего два диска, вторая же задача может рассматриваться как элементарная, так как ее решение состоит ровно из одного хода — перемещения диска 3 на колышек *C*. В первой и третьей задачах можно вновь применить метод редукции задач, и свести их к элементарным задачам. Весь процесс редукции можно схематически представить в виде дерева на рис. 9. Вершины дерева соответствуют решаемым задачам/подзадачам, причем листья дерева соответствуют элементарным задачам перемещения дисков, а дуги связывают редуцируемую задачу с ее подзадачами.

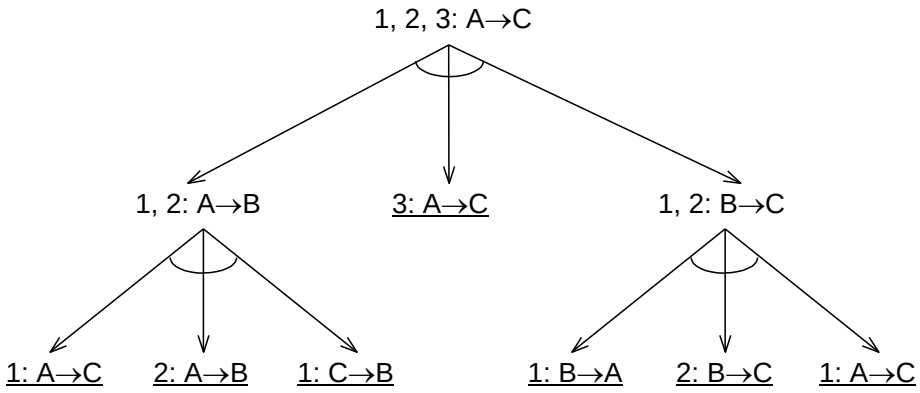

Рис. 9. Редукция задачи о ханойской башне

Заметим, что рассмотренная идея сведения задачи к совокупности подзадач может быть применена и в случае, когда начальная конфигурация задачи о пирамидке содержит не три, а большее число дисков. Таким образом, в случае подхода, основанного на редукции задач, мы получаем также пространство, но состоящее не из состояний, а из задач/подзадач (точнее, их описаний). При этом роль, аналогичную операторам в пространстве состояний, играют операторы, сводящие задачи в подзадачи. Точнее, каждый ***оператор редукции*** преобразует описание задачи в описание множества подзадач, причем

это множество таково, что решение всех подзадач обеспечивает решение редуцированной задачи.

При решении задач методом редукции, как и при решении в пространстве состояний, может возникнуть необходимость перебора. Действительно, на каждом этапе редукции может оказаться несколько применимых операторов (т.е. способов сведения задачи к подзадачам) и, соответственно, несколько альтернативных множеств подзадач. Некоторые способы, возможно, не приведут к решению исходной задачи, поскольку могут обнаружиться неразрешимые подзадачи, другие же способы могут дать окончательное решение. В общем случае для полной редукции исходной задачи необходимо перепробовать несколько операторов. Процесс редукции продолжается, пока исходная задача не будет сведена к набору элементарных задач, решение которых известно.

Аналогично представлению в пространстве состояний, формализация задачи в рамках подхода, основанного на редукции задач, включает определение следующих составляющих:

- формы описания задач/подзадач и описание исходной задачи;
- множества операторов и их воздействий на описания задач;
- множества элементарных задач.

Эти составляющие задают неявно ***пространство задач***, в котором требуется провести поиск решения задачи.

Что касается формы описания задач/подзадач, то часто их удобно описывать в терминах пространства состояний, т.е. задавая начальное состояние и множество операторов, а также целевое состояние или его свойства. В этом случае элементарными задачами могут быть, к примеру, задачи, решаемые за один шаг в пространстве состояний.

Если выбрать такую форму описания в задаче о пирамидке, то элементарная задача перекладывания самого большого диска с колышка *A* на *C* записывалась бы как (*BBA*)⇨(*BBC*), а исходная задача — как (*AAA*)⇨(*CCC*). Причем, поскольку множество операторов предполагается одним и тем же для всех задач/подзадач, то в их описаниях оно в явном виде может не фигурировать. При выбранной форме описания задач результирующие подзадачи естественно интерпретируются как задачи нахождения пути между определенными состояниями-вехами в пространстве состояний. К примеру, такими вехами являются состояния (*BBA*) и (*BBC*) задачи о пирамидке, изображенные на рис.8. Они обладают тем свойством, что через них пройдет и искомый решающий путь.

В дополнение отметим, что подход с использованием пространства состояний можно рассматривать как вырожденный случай подхода, основанного на редукции задач, так как применение оператора в пространстве состояний сводит обычно исходную задачу к несколько более простой задаче, т.е. редуцирует ее. При этом результирующее множество подзадач состоит только из одного элемента, т.е. мы имеем простейший случай замены редуцируемой задачи на ей эквивалентную.

1.2.2 И/ИЛИ графы. Решающий граф

Для изображения процесса редукции задач и получающихся при этом альтернативных множеств подзадач используются обычно графоподобные структуры, вершины которых представляют описания задач и подзадач, а каждая дуга связывает пару вершин, соответствующих редуцируемой задаче и одной из результирующих подзадач, причем стрелки на дугах указывают направление редукции. Пример такой структуры приведен на рис.10(а): задача *G* может быть решена путем решения либо задач D_1 и D_2, либо E_1, E_2 и E_3 либо задачи *F*. При этом ребра, относящиеся к одному и тому же множеству подзадач, связываются специальной дугой. Чтобы сделать такую структуру более наглядной, вводятся дополнительные промежуточные вершины, и каждое множество результирующих задач группируется под своей родительской вершиной. При этом структура на рис.10(а) преобразуется в структуру, изображенную на рис.10(б): для двух из трех альтернативных множеств подзадач добавлены соответственно вершины *D* и *E*.

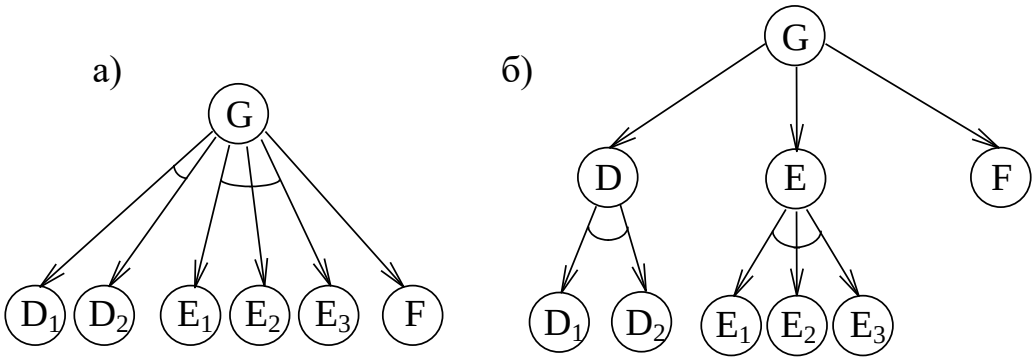

Рис. 10. Схематическое изображение редукции задач

Если считать, что вершины *D* и *E* соответствуют описаниям альтернативных путей решения исходной задачи, то вершину G можно назвать ***ИЛИ-вершиной***, так как задача *G* разрешима или способом *D*, или способом *E*, или способом *F*. Аналогично вершины *D* и *E* можно назвать ***И-вершинами***, поскольку каждый из соответствующих им способов требует решения всех подчиненных задач, что и обозначается специальной дугой. По этой причине структуры, подобные структурам, изображенным на рис.10(б) и рис.11, называются ***И/ИЛИ-графами*** (или ***графами целей***).

Если некоторая вершина такого графа имеет непосредственно следующие за ней (дочерние) вершины, то либо все они являются И-вершинами, либо все они — ИЛИ-вершины. Заметим, что если у некоторой вершины И/ИЛИ-графа имеется ровно одна дочерняя вершина, то последнюю можно считать как И-вершиной, так и ИЛИ-вершиной – такова, например, вершина *F* на рис.10(б).

На языке И/ИЛИ-графов применение нескольких операторов редукции задачи будет означать, что сначала будет построена промежуточная И-вершина, а затем непосредственно следующие за ней ИЛИ-вершины подзадач. Исключение составляет случай, когда множество задач состоит только из одного

элемента, в этом случае будет образована ровно одна вершина, будем для определенности считать ее ИЛИ-вершиной.

Вершину И/ИЛИ-графа, соответствующую описанию исходной задачи, будем называть ***начальной*** вершиной. Вершины же, которые соответствуют описаниям элементарных задач, будем называть ***заключительными*** вершинами. В графе, показанном на рис.11, начальной является вершина P_0, а заключительными — вершины P_1, P_4, P_5, P_7 и P_8 (они изображены жирными кружками).

Поиск решения задачи, осуществляемый путем перебора вершин графа, применим и в подходе, основанном на редукции задач. Цель поиска на И/ИЛИ-графе — показать, что ***разрешима*** исходная задача, т.е. начальная вершина. Разрешимость этой вершины зависит от разрешимости других вершин графа.

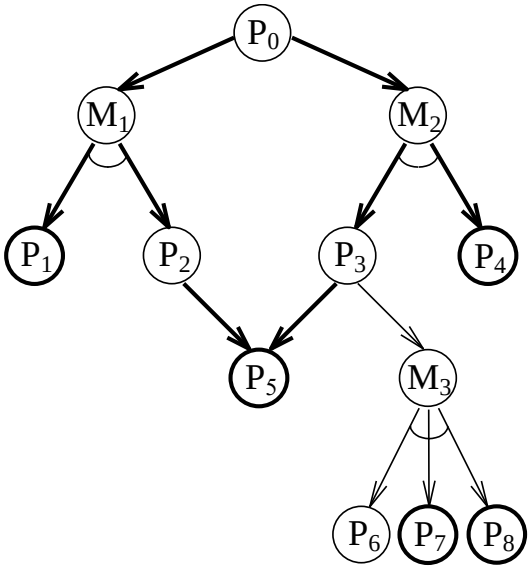

Рис. 11. Пример И/ИЛИ-графа

Сформулируем общее рекурсивное определение разрешимости вершины в И/ИЛИ-графе:

- заключительные вершины разрешимы, так как они соответствуют элементарным задачам;
- ИЛИ-вершина, не являющаяся заключительной, разрешима тогда и только тогда, когда разрешима по крайней мере одна из ее дочерних вершин;
- И-вершина, не являющаяся заключительной, разрешима тогда и только тогда, когда разрешима каждая из ее дочерних вершин.

Если в процессе поиска удалось показать, что начальная вершина разрешима, то это значит, что обнаружено решение исходной задачи, которое заключено в так называемом решающем графе. ***Решающий граф*** — это подграф И/ИЛИ-графа, состоящий только из разрешимых вершин и доказывающий разрешимость начальной вершины.

Для И/ИЛИ-графа, изображенного на рис.11, разрешимыми являются (кроме заключительных) вершины M_1, M_2, P_2, P_3. Этот граф содержит два

решающих графа: первый состоит из вершин P_0, M_1, P_1, P_2 и P_5 ; а второй — из вершин P_0, M_2, P_3, P_4 и P_5. Заметим, что вершины M_3 и P_6 не являются разрешимыми (M_3 неразрешима в силу неразрешимости вершины P_6).

1.2.3 Пример: задача символьного интегрирования

В качестве примера применения метода редукции рассмотрим решение задачи символьного интегрирования, т.е. нахождения неопределенного интеграла $\int F(x)dx$. Обычно эта задача решается путем последовательного преобразования интеграла к выражению, содержащему известные табличные интегралы. Для этого используется несколько правил интегрирования, в том числе: правило интегрирования суммы функций, правило интегрирования по частям, правило вынесения постоянного множителя за знак интегрирования, а также применение алгебраических и тригонометрических подстановок и использование различных алгебраических и тригонометрических тождеств.

Для формализации этой задачи в рамках подхода, основанного на редукции задач, необходимо определить форму описания задач/подзадач, операторы редукции и элементарные задачи. В качестве возможной формы описания задач может быть взята символьная строка, содержащая запись подынтегральной функции и переменной интегрирования (если последняя не фиксирована заранее). Операторы редукции будут основаны, очевидно, на упомянутых правилах интегрирования. Например, правило интегрирования по частям

$$\int u dx = u\int dv - \int v du$$

сводит исходную задачу (неопределенный интеграл в левой части равенства) к двум подзадачам интегрирования (два соответствующих интеграла в правой части равенства).

Заметим, что часть получаемых таким образом операторов редукции (как, например, операторы, соответствующие правилу интегрирования по частям или правилу интегрирования суммы функций), действительно редуцируют исходную задачу и порождают И-вершину в И/ИЛИ-графе, в то время как алгебраические и тригонометрические подстановки и тождества (как, например, деление числителя на знаменатель или дополнение до полного квадрата) лишь заменяют одно подынтегральное выражение на другое, порождая, таким образом, ИЛИ-вершины.

Элементарные задачи интегрирования соответствуют табличным интегралам, например:

$$\int \sin x \, dx = -\cos x + C$$

Отметим, что поскольку каждая из таких табличных формул содержит переменные, на самом деле она является схемой, задающей бесконечное множество элементарных задач.

Особенностью задачи интегрирования является то, что, например, при интегрировании по частям может оказаться несколько способов разбиения исходного подынтегрального выражения на части и соответственно несколько способов применения этого правила интегрирования. Это означает, что в общем случае для одного правила возможно несколько вариантов редукции задачи, т.е. несколько способов применения одного и того же оператора редукции.

Другая особенность рассматриваемой задачи состоит в том, что на каждом шаге редукции применимо обычно большое количество операторов (включая несколько применений одного и того же оператора), и получающийся И/ИЛИ-граф задачи слишком велик даже для несложных задач интегрирования. Поэтому, чтобы сделать поиск на таком графе достаточно эффективным, необходимо как-то ограничивать и/или упорядочивать множество порождаемых при поиске вершин. К примеру, можно упорядочить операторы редукции по степени их полезности, и приписать больший приоритет операторам, соответствующим правилам интегрирования суммы и интегрирования по частям.

На рис.12 показан решающий граф для одной задачи интегрирования. В вершинах графа указаны соответствующие задачи/подзадачи, заключительные вершины заключены в двойные рамки. Решение задачи может быть собрано из содержащейся в графе информации. Оно состоит из следующих шагов:

1. применения подстановки $z = tg\,(x)$;
2. эквивалентного преобразования подынтегрального выражения;
3. применения правила интегрирования суммы функций, причем одна из трех результирующих подзадач оказывается элементарной, а две остальные решаются за один шаг (первая — путем вынесения постоянного множителя за знак интеграла, вторая — применением подстановки $z = tg\,(w)$).

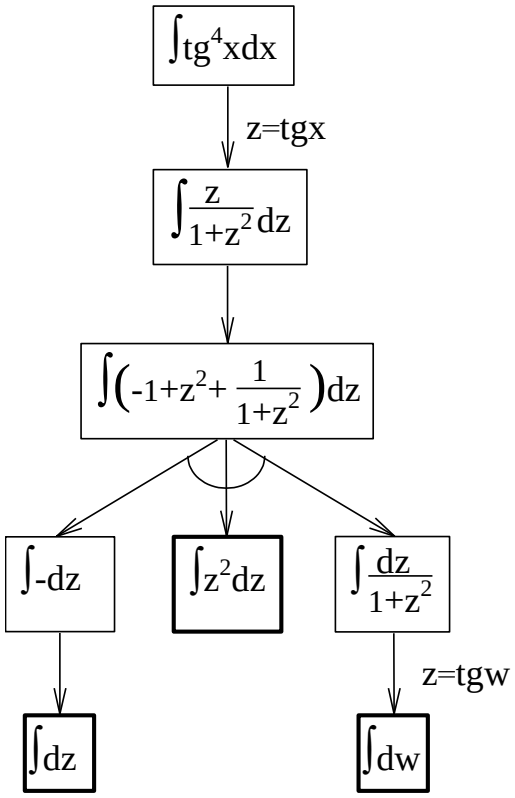

Рис. 12. Решающий граф для одной задачи интегрирования

Подход, основанный на редукции задач, применим и имеет преимущества по сравнению с подходом, использующим представление в пространстве состояний, когда получающиеся при редукции подзадачи можно решать независимо друг от друга, как в примере с интегрированием. Впрочем, это условие взаимной независимости результирующих задач можно несколько ослабить. Метод редукции применим также, если решения получающих подзадач зависят друг от друга, но при этом существует такой порядок их редукции, при котором найденные решения одних, более ранних подзадач не разрушаются при решении других, более поздних — как в головоломке о ханойской башне, в которой важен лишь порядок решения выделяемых подзадач.

ГЛАВА 2. АЛГОРИТМЫ ПОИСКА РЕШЕНИЯ

Оба рассмотренных подхода к решению задач – представление в пространстве состояний и метод редукции задач – предполагают поиск на графе соответствующего вида. Настоящая глава будет в основном посвящена алгоритмам поиска решения в пространстве состояний, для алгоритмов поиска на И/ИЛИ-графах рассматриваются только их отличительные особенности.

2.1 Разновидности поиска

Как уже отмечалось, поиск в пространстве состояний базируется на последовательном построении (переборе) вершин графа состояний — до тех пор, пока не будет обнаружено целевое состояние. Введем несколько терминов, которые будем использовать для описания различных алгоритмов поиска.

Вершину графа, соответствующую начальному состоянию, естественно назвать ***начальной*** вершиной, а вершину, соответствующую целевому состоянию – ***целевой***. Как и ранее, вершины, непосредственно следующие за некоторой вершиной, т.е. получившиеся в результате применения к ней допустимых операторов, будем называть *дочерними*, а саму исходную вершину — *родительской*. Основной операцией, выполняемой при поиске на графе, будем считать ***раскрытие вершины***, что означает порождение (построение) всех ее дочерних вершин, путем применения к соответствующему описанию состояния задачи всех допустимых операторов.

Поиск в пространстве состояний можно представить как процесс постепенного раскрытия вершин и проверки свойств порождаемых вершин. Важно, что в ходе этого процесса должны сохраняться *указатели* от всех возникающих дочерних вершин к их родительским. Именно эти указатели позволят восстановить путь назад к начальной вершине после того, как будет построена целевая вершина. Этот путь, взятый в обратном направлении, точнее, последовательность операторов, соответствующих дугам этого пути, и будет искомым решением задачи.

Вершины и указатели, построенные в процессе поиска, образуют поддерево всего неявно определенного при формализации задачи графа-пространства состояний. Это поддерево называется ***деревом поиска***.

Процедуры поиска в пространстве состояний различаются несколькими характеристиками, основными из которых являются:

- использование эвристической информации;
- порядок раскрытия (перебора) вершин;
- полнота просмотра пространства состояний;
- направление поиска.

В соответствии с первой характеристикой различают ***слепой*** и ***эвристический*** поиск. В первом случае местонахождение целевой вершины в пространстве состояний никак не влияет на порядок, в котором раскрываются

(перебираются) вершины. В противоположность слепому поиску, эвристический поиск использует априорную, эвристическую информацию об общем виде пространства состояний и/или о том, где в этом пространстве расположена цель, поэтому для раскрытия обычно выбирается наиболее перспективная вершина. В общем случае это позволяет сократить возникающий перебор вершин.

Два основных подвида слепого поиска, различающиеся порядком раскрытия вершин, — это ***поиск вширь*** и ***поиск вглубь***.

Как слепой, так и эвристический поиск могут характеризоваться полнотой просмотра пространства состояний. При ***полном*** переборе осуществляется исчерпывающий просмотр графа-пространства состояний, тем самым гарантируется нахождение решения, если таковое существует. В случае ***неполного*** поиска просматривается лишь некоторая, хотя и существенная часть пространства, и если эта часть не содержит целевых вершин, то искомое решение задачи найдено не будет.

В соответствии с направлением различают ***прямой*** поиск, ведущийся от начальной вершины к целевой, ***обратный*** поиск, ведущийся в направлении от целевой вершины к начальной, и ***двунаправленный*** поиск, при котором чередуются прямой и обратный поиск. Обратный поиск возможен в случае обратимости операторов задачи. Чаще используются (отчасти, в силу их простоты) алгоритмы прямого поиска, они и рассматриваются в данном пособии.

2.2 Слепой поиск

Слепые алгоритмы поиска вширь (*breadth_first_search*) и поиска вглубь (*depth_first_search*) различаются тем, какая вершина выбирается для очередного раскрытия. В алгоритме перебора вширь вершины раскрываются в том порядке, в котором они строятся. В алгоритме же перебора вглубь прежде всего раскрываются те вершины, которые были построены последними.

Сначала рассмотрим эти алгоритмы для пространств, являющихся деревьями (корнем дерева является начальная вершина). Затем покажем, как алгоритмы следует модифицировать для поиска на произвольных графах. Организовать перебор в деревьях проще, так как при построении нового состояния (и соответствующей вершины) можно быть уверенным в том, что такое состояние никогда раньше не строилось и не будет строиться в дальнейшем.

Описываемые алгоритмы используют списки *Open* и *Closed*, состоящие соответственно из раскрытых и нераскрытых вершин пространства состояний. Имя *Current* обозначает вершину, раскрываемую в текущий момент поиска. При формулировке алгоритмов предполагаем, что начальная вершина не является целевой.

2.2.1 Поиск вширь

Базовый **алгоритм поиска вширь** состоит из следующей последовательности шагов:

1. **Шаг 1.** Поместить начальную вершину в список нераскрытых вершин *Open*.
2. **Шаг 2.** Если список *Open* пуст, то завершить поиск и выдать сообщение о неудаче, в противном случае перейти к следующему шагу.
3. **Шаг 3.** Выбрать первую вершину из списка *Open* (назовем ее *Current*) и перенести ее в список раскрытых вершин *Closed*.
4. **Шаг 4.** Раскрыть вершину *Current*, образовав все ее дочерние вершины. Если дочерних вершин нет, то перейти к шагу 2, иначе поместить все дочерние вершины (в любом порядке) в конец списка *Open* и построить указатели, ведущие от этих вершин к родительской вершине *Current*.
5. **Шаг 5.** Проверить, нет ли среди дочерних вершин целевых. Если есть хотя бы одна целевая вершина, то завершить поиск и выдать решение задачи, получающееся просмотром указателей назад от найденной целевой вершины к начальной. В противном случае перейти к шагу 2.
6. Конец алгоритма.

Основу этого алгоритма составляет цикл последовательного раскрытия (шаги 2-5) концевых вершин (листьев) дерева перебора, хранящихся в списке *Open*. Описанный алгоритм поиска вширь является полным. Можно также показать, что при переборе вширь непременно будет найден самый короткий путь к целевой вершине – при условии, что этот путь вообще существует. Если же решающего пути нет, то в случае конечного пространства состояний алгоритм завершит работу (с сообщением о неудаче поиска), в случае же бесконечного пространства алгоритм не остановится.

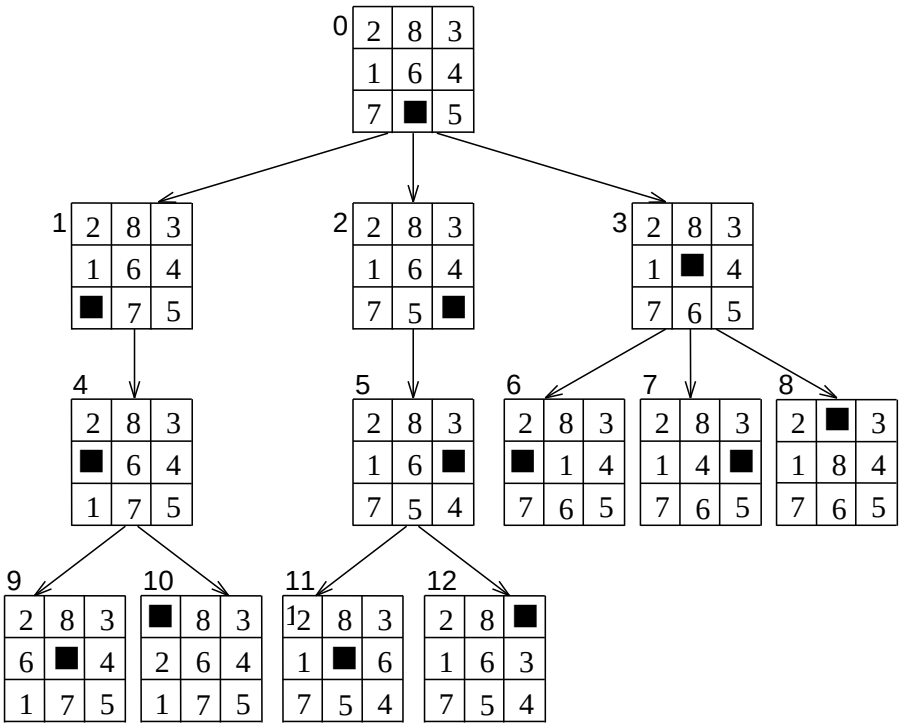

Рис.13. Поиск вширь для игры в восемь

На рис.13 показана часть дерева, построенного в результате применения алгоритма поиска вширь к некоторой начальной конфигурации игры в восемь, причем выполнение алгоритма прервано после построения первых 12 вершин (при этом раскрыто 6 вершин). В вершинах дерева помещены соответствующие описания состояний. Эти вершины перенумерованы в том порядке, в котором они были построены в ходе поиска. На следующем шаге цикла алгоритма будет раскрываться одна из вершин с номерами 6, 7 или 8, поскольку они расположены в начале списка нераскрытых вершин.

Считаем, что порядок построения дочерних вершин соответствует следующему зафиксированному порядку перемещения пустой клетки («пустышки»): влево/вправо/вверх/вниз. Предполагается также, что используемая алгоритмом операция раскрытия вершин организована таким образом, что она не порождает никакое состояние, идентичное состоянию в уже построенной вершине, являющейся родительской для раскрываемой вершины. Тем самым в дереве перебора нет дублирования одного и того же состояния в вершинах, имеющих общую соседнюю вершину.

В приведенном примере алгоритм поиска вглубь, сформулированный для пространств, являющихся деревьями, применялся к пространству состояний, являющемуся графом (в котором могут быть циклы). В некоторых случаях это допустимо, т.е. алгоритм находит решение, если оно есть, и заканчивает работу. Причем и при поиске в пространствах-деревьях, и при поиске в пространствах-графах, построенная алгоритмом структура из вершин и указателей всегда образует дерево (дерево перебора), поскольку указатели от дочерних вершин ссылаются только на одну порождающую вершину. Но в случае поиска на произвольном графе (и в этом – отличие от деревьев-пространств) одно и тоже

состояние может быть продублировано в разных частях построенного дерева поиска: граф как бы разворачивается в дерево путем дублирования некоторых его частей. Например, по принятому предположению об операции раскрытия в игре в восемь исключалось только повторное возникновение состояний, уже встречавшихся два шага вверх по дереву перебора, другие же, более далекие друг от друга повторы одного и того же состояния остаются возможными. В общем случае, вследствие многократного дублирования вершин (из-за циклов в графе) возможно зацикливание базового алгоритма поиска вширь.

2.2.2 Поиск вглубь

Для формулировки алгоритма поиска вглубь необходимо определить понятие ***глубины вершины*** в дереве поиска. Это можно сделать следующим образом:

- глубина *корня* дерева равна нулю;
- глубина каждой *некорневой* вершины на единицу больше глубины ее родительской вершины.

В алгоритме перебора вглубь раскрытию в первую очередь подлежит вершина, имеющая наибольшую глубину. Такой принцип может привести к не завершающемуся процессу — это происходит, если пространство состояний бесконечно, и поиск вглубь пошел по бесконечной ветви дерева, не содержащей целевой вершины. Поэтому необходимо то или иное ограничение поиска, самый распространенный способ — ограничить глубину просмотра дерева. Это означает, что в ходе перебора можно строить только такие вершины, глубина которых не превышает некоторую заданную ***граничную глубину***. Тем самым, раскрытию в первую очередь подлежит вершина наибольшей глубины, но только если она расположена выше фиксированной границы. Соответствующий алгоритм поиска называется ***ограниченным поиском вглубь***.

Основные шаги базового алгоритма ограниченного поиска вглубь (с граничной глубиной *D*) таковы:

1. **Шаг 1.** Поместить начальную вершину в список нераскрытых вершин *Open*.
2. **Шаг 2.** Если список *Open* пуст, то завершить поиск и выдать сообщение о неудаче, в противном случае перейти к следующему шагу.
3. **Шаг 3.** Выбрать первую вершину из списка *Open* (назовем ее *Current*) и перенести ее в список раскрытых вершин *Closed*.
4. **Шаг 4.** Если глубина вершины *Current* равна граничной глубине *D*, то перейти к шагу 2, в ином случае перейти к следующему шагу.
5. **Шаг 5.** Раскрыть вершину *Current*, построив все ее дочерние вершины. Если дочерних вершин нет, то перейти к шагу 2, иначе поместить все дочерние вершины (в произвольном порядке) в начало списка *Open* и построить указатели, ведущие от этих вершин к родительской вершине *Current*.
6. **Шаг 6.** Если среди дочерних есть хотя бы одна целевая вершина, то завершить поиск и выдать решение задачи, получающееся просмотром

назад указателей от найденной целевой вершины к начальной. В противном случае перейти к шагу 2.

7. Конец алгоритма.

Приведенное описание очень похоже на описание алгоритма поиска вширь, разница заключается только в ограничении глубины (шаг 4) и в участке списка *Open*, куда помещаются построенные дочерние вершины (шаг 5).

Поскольку глубина поиска ограничена, то будучи примененным к деревьям-пространствам состояний, описанный базовый алгоритм поиска вглубь всегда заканчивает работу. В отличие от алгоритма поиска вширь, он осуществляет неполный поиск, поскольку целевая вершина может располагаться ниже граничной глубины (в таком случае она не будет обнаружена).

На рис.14 показана часть дерева перебора, построенного алгоритмом поиска вглубь при граничной глубине равной 4. В качестве начального состояния взята та же самая конфигурация игры в восемь, что и в примере на рис.13. В ходе поиска также построено 12 вершин (раскрыто 7), вершины перенумерованы в том порядке, в котором они были построены. Как нетрудно убедиться, сравнивая два указанных рисунка, алгоритмами поиска вширь и вглубь построены разные деревья поиска. Видно, что в алгоритме поиска в глубину сначала идет поиск вдоль одного пути, пока не будет достигнута установленная граничная глубина, затем рассматриваются альтернативные пути той же или меньшей глубины, которые отличаются от первого пути лишь последней (концевой) вершиной, после чего рассматриваются пути, отличающиеся последними двумя вершинами, и т.д.

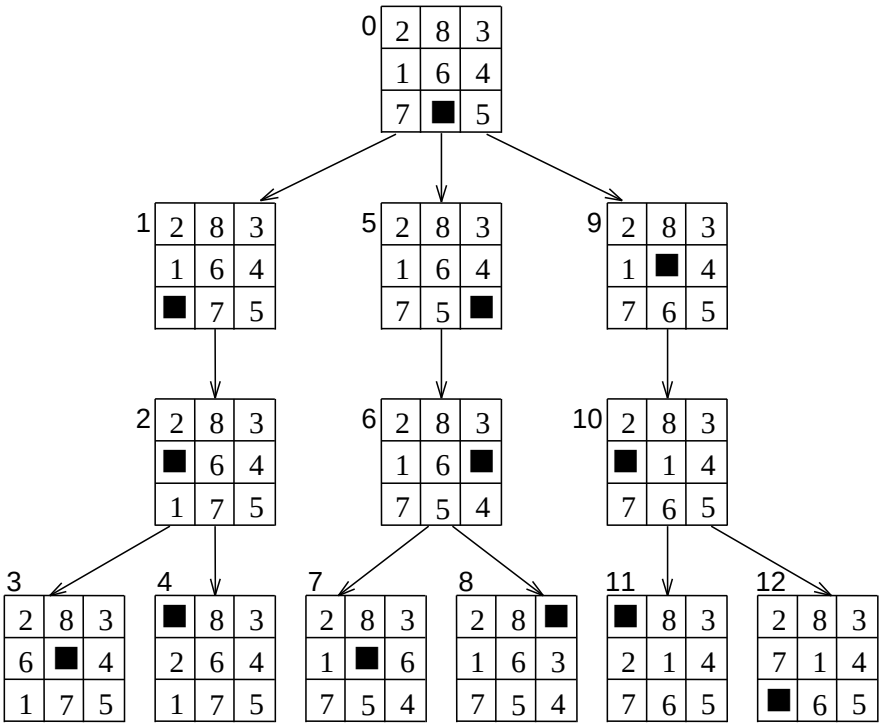

Рис.14. Поиск вглубь для игры в восемь

2.2.3 Слепой перебор и бектрекинг

Если продолжить выполнение алгоритмов перебора вширь и вглубь для рассмотренного на рис.13 и 14 начального состояния игры в восемь, с целью поиска конфигурации, заданной на рис.1(б), то она будет найдена на глубине 5, при этом алгоритмом поиска вширь будет раскрыто 26 и построено 46 вершин, а алгоритмом поиска вглубь – соответственно 18 и 35 вершин (см. [Нильсон73, раздел 3.2] или [Нильсон85, раздел 2.4]).

В целом, алгоритмы поиска вширь и вглубь сравнимы по эффективности (в частности, по количеству построенных вершин). В то же время алгоритм поиска вглубь может оказаться предпочтительнее – в тех случаях, когда он начат с ветви дерева, содержащей целевое состояние, решение задачи будет обнаружено раньше, чем при поиске вширь.

Подчеркнем, что, так же как и при переборе вширь, при переборе вглубь формируется именно дерево, а не граф поиска, даже если пространство состояний представлялось графом с циклами. В последнем случае, однако, дерево перебора может содержать дубликаты состояний и возможно зацикливание алгоритма. Если, к примеру, в графе две вершины являются друг для друга дочерними, то они будут многократно дублироваться в списке *Open*, что приводит к зацикливанию.

Чтобы избежать такого дублирования вершин в случае перебора на графах общего вида, необходимо внести некоторые очевидные изменения в описанные базовые алгоритмы поиска вширь и вглубь.

В алгоритме перебора вширь следует дополнительно проверять, не находится ли каждая вновь построенная вершина (точнее, соответствующее описание состояния) в списках *Open* и *Closed* по той причине, что она уже строилась раньше в результате раскрытия какой-то другой вершины. Если это так, то такую вершину не надо снова помещать в список *Open* (таким образом, разрывается цикл графа-пространства и обрывается соответствующая ветвь дерева перебора). В алгоритме же ограниченного поиска вглубь, кроме указанной проверки, может оказаться необходимым пересчет глубины порожденной дочерней вершины, уже имеющейся либо в списке *Open*, либо в списке *Closed*.

Усовершенствованный таким образом алгоритм поиска вширь всегда завершит работу в случае существования решения, а усовершенствованный алгоритм поиска вглубь закончится в любом случае, независимо от существования решения. В разделе 2.4 настоящей главы приводится реализация этих алгоритмов на языке программирования Лисп [Семенов]. Немаловажно, что алгоритмы слепого перебора описаны нами в форме, пригодной для их программирования с использованием любого языка, не только языка программирования задач искусственного интеллекта.

Алгоритм поиска вглубь демонстрирует также способ решения поисковых задач, называемый ***бектрекингом*** (*backtracking*). Этот способ предлагает

определенную организацию перебора всех возможных вариантов решения задачи, число которых может быть велико.

Суть бектрекинга состоит в том, чтобы в каждой точке процесса решения задачи, где существует несколько априори равноправных альтернативных вариантов (путей) дальнейшего продолжения, выбрать один из них и следовать ему, предварительно запомнив другие альтернативы – для того, чтобы в случае неуспешности выбранного варианта-пути вернуться в точку выбора и выбрать для продолжения решения другой возможный вариант-путь. В общем случае в процессе решения возможно возникновение многих подобных точек выбора, называемых обычно *точками бектрекинга* или *развилками*; к каждой из таких точек может потребоваться *возврат* для выбора других вариантов решения.

В базовом алгоритме поиска вглубь по существу проводится бектрекинг. Действительно, запоминание всех альтернатив продолжения поиска (нераскрытых вершин) осуществляется в списке *Open*, на шаге 3 производится выбор варианта-альтернативы, а возврат к этому шагу для выбора следующей альтернативы осуществляется на шагах 4 и 5.

Некоторые языки для задач искусственного интеллекта, как, например, Пролог [Братко] и Плэнер [Пильщиков] имеют специальный встроенный механизм для реализации бектрекинга. Это означает, что запоминание точек бектрекинга – самих альтернатив и связанной с ними информации, а также реализация возвратов к нужным точкам (с восстановлением всей операционной обстановки в этой точке) возложены на интерпретатор языка, т.е. делается автоматически. От программиста требуется лишь определение точек бектрекинга с нужными альтернативами и инициация в необходимый момент процесса возврата. Заметим попутно, что язык Плэнер, по сравнению с Прологом, предлагает более гибкие средства управления бектрекингом, краткое описание этих средств находится в разделе 4.2 пособия.

Встроенный механизм бектрекинга позволяет упростить разработку поисковой программы, укорачивая ее текст – это демонстрирует приведенная в разделе 2.4 плэнерская функция *DEPTH_FIRST_SEARCH*, реализующая поиск вглубь (без ограничения глубины).

В целом алгоритмы слепого перебора являются неэффективными методами поиска решения, и в случае нетривиальных задач их невозможно использовать из-за большого числа порождаемых вершин. Действительно, если L — длина решающего пути, а B — количество ветвей (дочерних вершин) у каждой вершины, то в общем случае для нахождения решения надо исследовать BL путей, ведущих из начальной вершины. Эта величина растет экспоненциально с ростом длины решающего пути, что приводит к ситуации, называемой комбинаторным взрывом.

Один из способов повышения эффективности поиска связан с использованием информации, отражающей специфику решаемой задачи и позволяющей более целенаправленно двигаться к цели. Такая информация обычно называется эвристической, а соответствующие алгоритмы и методы поиска — эвристическими.

2.3 Эвристический поиск

Идея, лежащая в основе эвристического поиска, состоит в том, чтобы с помощью эвристической информации оценивать перспективность нераскрытых вершин пространства состояний (с точки зрения достижения цели) и выбирать для продолжения поиска наиболее перспективную вершину.

Самый распространенный способ использования эвристической информации — введение так называемой ***эвристической оценочной функции***. Эта функция определяется на множестве вершин пространства состояний и принимает числовые значения. Значение эвристической оценочной функции *Est*(*V*) может интерпретироваться как перспективность раскрытия вершины (иногда — как вероятность ее расположения на решающем пути). Обычно считают, что меньшее значение *Est*(*V*) соответствует более перспективной вершине, и вершины раскрываются в порядке увеличения (точнее, неубывания) значения оценочной функции.

2.3.1 Алгоритм эвристического поиска

Последовательность шагов формулируемого ниже базового ***алгоритма эвристического (упорядоченного) перебора*** похожа на последовательность шагов алгоритмов слепого перебора, отличие заключается в использовании эвристической оценочной функции. После порождения нового состояния производится его оценивание (т.е. вычисление значения этой функции). Списки открытых и закрытых вершин содержат как вершины, так и их оценки, которые и используются для упорядочения поиска.

В цикле каждый раз для раскрытия выбирается наиболее перспективная концевая вершина дерева перебора. Как и в случае алгоритмов слепого поиска, множество порождаемых алгоритмом вершин и указателей образует дерево, в листьях которого находятся нераскрытые вершины.

Предполагаем, что исследуемое алгоритмом пространство состояний представляет собой дерево. Укажем основные шаги базового алгоритма эвристического поиска (*best_first_search*):

1. **Шаг 1.** Поместить начальную вершину в список нераскрытых вершин *Open* и вычислить ее оценку.
2. **Шаг 2.** Если список *Open* пуст, то завершить поиск и выдать сообщение о неудаче, в противном случае перейти к шагу 3.
3. **Шаг 3.** Выбрать из списка *Open* вершину с минимальной оценкой (среди вершин с одинаковой минимальной оценкой выбирается любая) и перенести эту вершину (назовем ее *Current)* в список *Closed*.
4. **Шаг 4.** Если *Current* – целевая вершина, то завершить поиск и выдать решение задачи, получающееся просмотром указателей от нее к начальной вершине, в противном случае перейти к шагу 5.
5. **Шаг 5.** Раскрыть вершину *Current*, построив все ее дочерние вершины. Если таких вершин нет, то перейти к шагу 2, в ином случае — к шагу 6.

6. **Шаг 6.** Для каждой дочерней вершины вычислить оценку (значение оценочной функции), поместить все дочерние вершины в список *Open* и построить указатели, ведущие от этих вершин к родительской вершине *Current*. Перейти к шагу 2.
7. Конец алгоритма.

Заметим, что поиск вглубь можно рассматривать как частный случай эвристического поиска с оценочной функцией $Est(V) = d(V)$, а поиск вширь — с оценочной функцией $Est(V) = 1/d(V)$, где $d(V)$ – глубина вершины V.

Чтобы модифицировать рассмотренный алгоритм для перебора на произвольных графах-пространствах состояний, необходимо предусмотреть в нем реакцию на случай построения дочерних вершин, которые уже имеются либо в списке раскрытых, либо в списке нераскрытых вершин.

В принципе, эвристическая оценочная функция может зависеть не только от внутренних свойств оцениваемого состояния (т.е. свойств входящих в описание состояния элементов), но и от характеристик всего пространства состояний, например, от глубины местонахождения оцениваемой вершины в дереве перебора или других свойств пути к этой вершине. Поэтому значение оценочной функции для вновь построенной дочерней вершины, входящей в список *Open* или *Closed*, может понизиться, и надо скорректировать старую оценку вершины, заменив ее на новую, меньшую. Если вновь построенная вершина с меньшей оценкой входит в список *Closed*, необходимо вновь поместить ее в список *Open*, но с меньшей оценкой. Потребуется также изменить направления указателей от всех вершин списков *Open* и *Closed*, оценка которых уменьшилась, направив их к вершине *Current*.

Впрочем, если оценочная функция учитывает только внутренние характеристики вершин-состояний, то для предотвращения зацикливания требуется более простая модификация алгоритма – надо просто исключить дублирование состояний в списках *Open* и *Closed*, оставляя в них лишь по одному состоянию (именно этот простой способ и реализован во всех лисп-алгоритмах поиска раздела 2.4).

Проиллюстрируем работу алгоритма эвристического поиска опять же на примере игры в восемь (для той же начальной ситуации, что на рис. 13 и 14, и целевого состояния, указанного на рис.1(б)). Воспользуемся в качестве оценочной следующей простой функцией:

$$Est1(V) = d(V) + k(V)$$

где:

- $d(V)$ — глубина вершины V, или число ребер дерева на пути от этой вершины к начальной вершине;
- $k(V)$ — число фишек позиции-вершины V, стоящих не на «своем» месте (фишка стоит не на «своем» месте, если ее позиция отлична от позиции в целевом состоянии).

На рис. 15 показано дерево, построенное алгоритмом эвристического перебора с использованием указанной оценочной функции. Оценка каждой вершины приведена рядом с ней внутри кружка. Отдельно стоящие цифры, как и раньше, показывают порядок, в котором строились вершины. Двойной рамкой обведена найденная целевая вершина, она построена двенадцатой.

Видно, что поскольку каждый раз выбор вершины с минимальной оценкой производится внутри всего построенного к текущему моменту дерева перебора, то раскрываемые друг за другом вершины могут располагаться в отдаленных друг от друга частях дерева. Применяемая оценочная функция такова, что при прочих равных преимущество имеет менее глубокая вершина.

Решение задачи длиною в пять ходов найдено в результате раскрытия 6 и построения 13 вершин — это существенно меньше, чем при использовании слепого перебора (соответствующие числа были: для поиска вширь — 26 и 46 вершин, для поиска вглубь — 18 и 35 вершин). Таким образом, использование эвристической информации приводит к существенному сокращению перебора.

Эффективность алгоритмов поиска может быть оценена при помощи такого показателя, как *целенаправленность*. Он вычисляется по формуле

$$P = L / N,$$

где:

L – длина найденного пути до цели (она равна глубине целевой вершины);

N – общее число вершин, построенных в ходе перебора.

Легко заметить, что $P = 1$, если строятся только вершины решающего пути, а в остальных случаях $P < 1$. Вообще, эта величина тем меньше, чем больше строится бесполезных вершин. Таким образом, этот критерий показывает, насколько дерево, построенное при переборе, вытянуто, а не кустисто. Для рассмотренных примеров работы алгоритмов для игры в восемь этот показатель равен соответственно: эвристический поиск — 5/13, поиск вширь — 5/46, поиск вглубь — 5/35,

Ясно, что алгоритм эвристического поиска с хорошо подобранной оценочной функцией находит решение задачи быстрее алгоритмов слепого перебора. Однако подбор удачной эвристической функции, существенно сокращающей поиск, — наиболее трудный момент при формализации задачи, часто подходящая оценочная функция выявляется только экспериментально.

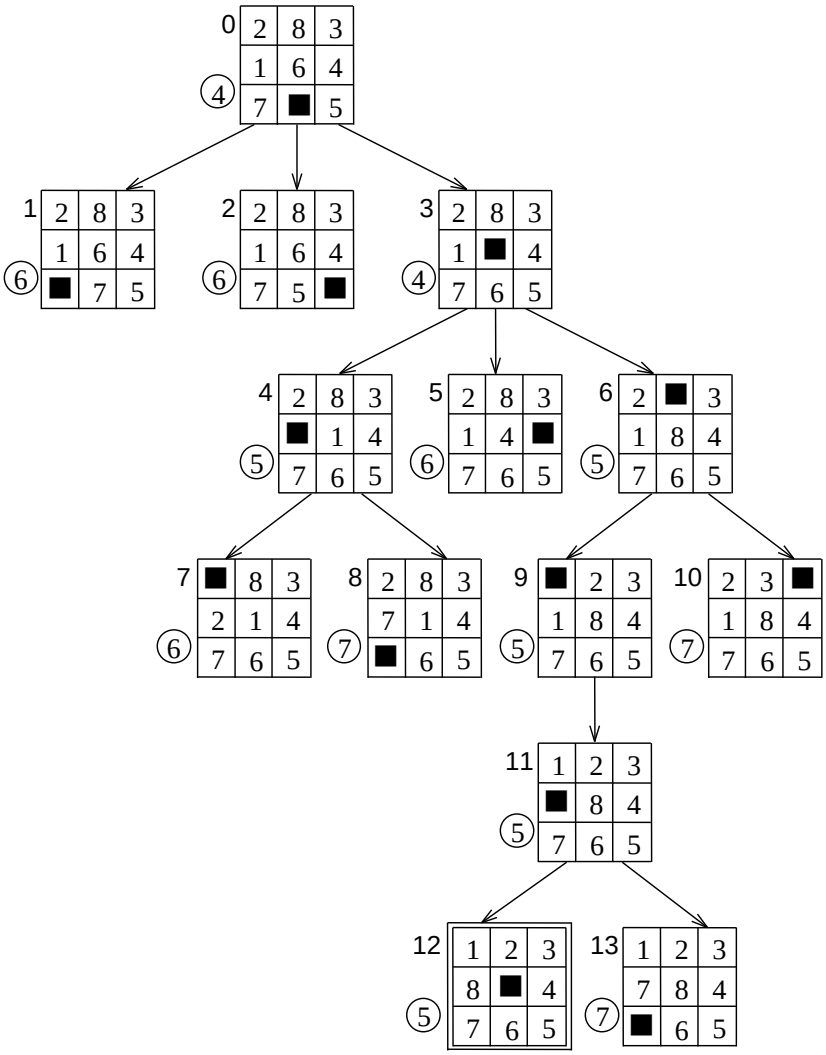

Рис. 15. Эвристический поиск для игры в восемь

В одной и той же задаче можно сравнивать различные оценочные функции по их *эвристической силе*, т.е. по тому, насколько они убыстряют поиск, делают его эффективным. Заметим, что эвристическая сила функции должна учитывать общий объем вычислительных затрат при поиске, поэтому кроме числа раскрытых и построенных вершин важен и такой показатель, как сложность вычисления самой оценочной функции.

Для игры в восемь можно предложить еще одну эвристическую функцию:

$$Est2(V) = d(V) + s(V).$$

Первое слагаемое $d(V)$ этой функции имеет тот же смысл, что и для функции *Est1*. Второе слагаемое получается подсчетом для каждой из восьми фишек суммы двух расстояний – по вертикали и горизонтали – между клетками, где находится фишка в оцениваемом и целевом состояниях, а затем вычислением общей суммы $s(V)$ таких расстояний для всех восьми фишек. Тем самым, $s(V)$ выражает «суммарное расстояние» всех фишек от их целевого положения.

Например, для начальной конфигурации на рис.15 расстояние текущего положения фишки с номером 8 от ее положения в целевой конфигурации равно

1 и по вертикали, и по горизонтали, а сумма их равна 2. Общая же сумма таких расстояний для всех фишек равна 5 (фишки 3, 4, 5, 7 стоят уже на «своем» месте, поэтому их вклад в суммарное расстояние равен 0). Интуитивно ясно, и это можно показать на примерах, что новая эвристическая функция имеет большую эвристическую силу, т.е. более эффективно направляет поиск к цели.

2.3.2 Допустимость алгоритма эвристического поиска

Важным является вопрос, может ли алгоритм эвристического перебора с оценочной функций общего вида (на которую не накладывается никаких ограничений) гарантировать нахождение решающего пути за конечное число шагов в тех случаях, когда решение существует (как это гарантирует алгоритм поиска вширь). Понятно, что такой уверенности нет, прежде всего, для задач с бесконечными пространствами состояний. Вообще же, нередка ситуация, когда эвристика, сильно сокращающая перебор для одних задач (начальных и целевых состояний), в то же время для других задач либо не уменьшает перебор (решение задачи может искаться даже дольше, чем с использованием слепого поиска), либо вовсе не обеспечивает обнаружение решающего пути.

Математическое исследование алгоритма эвристического поиска, прежде всего — условий, гарантирующих нахождение им решения, было проведено для эвристических оценочных функций специального вида и для более сложной задачи, чем рассматриваемая до сих пор задача поиска произвольного решающего пути до целевой вершины.

Предположим, что на множестве дуг пространства состояний определена функция стоимости:

c (V_A, V_B) – стоимость дуги-перехода от вершины V_A к вершине V_B.

Определим также ***стоимость*** любого пути в графе-пространстве как сумму стоимостей входящих в путь дуг. Пусть целью поиска будет не просто нахождение решающего пути, а нахождение ***оптимального решающего пути*** — решающего пути с минимальной стоимостью.

Предположим также, что эвристическая оценочная функция $Est(V)$ построена таким образом, чтобы оценивать стоимость оптимального решающего пути, идущего из начальной вершины к одной из целевой вершин, при условии, что этот путь проходит через вершину V. Тогда значение оценочной функции можно представить в виде суммы двух слагаемых:

$$Est(V) = g(V) + h(V) \qquad (*)$$

где:

- $g(V)$ — оценка оптимального пути от начальной вершины до вершины V,
- $h(V)$ — оценка оптимального пути от вершины V до целевой вершины.

Если в процессе поиска уже построена вершина V, то путь до нее найден, и его стоимость может быть вычислена. Найденный путь не обязательно

оптимален (возможно, существует более дешевый, еще не найденный путь из начальной вершины в V), однако стоимость найденного пути может быть использована в качестве оценки искомого пути минимальной стоимости из начальной вершины до V, т.е. в качестве первого слагаемого $g(V)$ эвристической функции. Второе же слагаемое $h(V)$ может быть предложено исходя из эвристических соображений, свойственных конкретной решаемой задаче, как некоторая характеристика-оценка текущей вершины V (близости ее к цели). Таким образом, собственно эвристическая информация будет воплощена только во втором слагаемом оценочной функции.

Разновидность алгоритма эвристического поиска, применяемого для поиска оптимального решающего пути и использующего при этом оценочную функцию указанного выше вида (*), известна в литературе как ***A-алгоритм*** [Нильсон85, раздел 2.4]. Были доказаны важные свойства этого алгоритма, прежде всего, утверждение о его допустимости.

Алгоритм перебора называют ***допустимым*** (или *состоятельным*), если для произвольного графа он всегда заканчивает свою работу построением оптимального пути к цели, при условии, что такой путь существует.

Пусть $h^*(V)$ — стоимость оптимального пути из произвольной вершины V в целевую вершину.

Верна следующая теорема о допустимости А-алгоритма:

А-алгоритм, использующий некоторую эвристическую функцию вида (*), где:

- $g(V)$ — стоимость пути от начальной вершины до вершины V в дереве перебора,
- $h(V)$ — эвристическая оценка оптимального пути из вершины V в целевую вершину, является допустимым, если $h(V) \le h^*(V)$ для всех вершин V пространства состояний.

А-алгоритм эвристического поиска с функцией $h(V)$, удовлетворяющей этому условию, получил название ***A*-алгоритма*** [Нильсон73, разделы 3.7-3.9; Лорьер, раздел 5.6].

Практическое значение этой теоремы в том, что для допустимости А-алгоритма достаточно найти какую-либо нижнюю грань функции $h^*(V)$ и использовать ее в качестве $h(V)$ — тогда оптимальность найденного алгоритмом решения будет гарантирована.

Если взять тривиальную нижнюю грань, т.е. установить $h(V) = 0$ для всех вершин пространства состояний, то допустимость будет обеспечена. Однако этот случай соответствует полному отсутствию какой-нибудь эвристической информации о задаче, и оценочная функция *Est* не имеет никакой эвристической силы, т.е. не сокращает возникающий перебор. А*-алгоритм ведет себя при этом аналогично алгоритму поиска вширь.

Точнее, при $Est(V) = g(V)$, где $g(V)$ — стоимость пути от начальной вершины до вершины V, мы получаем алгоритм, известный как ***алгоритм равных цен***. Алгоритм равных цен представляет собой более общий вариант метода перебора вширь, при котором вершины раскрываются в порядке возрастания

стоимости $g(V)$, т.е. в первую очередь раскрывается вершина из списка нераскрытых вершин, для которой величина g имеет наименьшее значение.

Если же, кроме того, положить стоимость c (V_A,V_B) = 0 для всех дуг пространства состояний, то А*-алгоритм просто превращается в неэффективный слепой поиск вширь.

Обе предложенные для игры в восемь эвристические функции $Est1(V)$ и $Est2(V)$ удовлетворяют условию допустимости А*-алгоритма. Первое их слагаемое $d(V)$ есть стоимость пути к вершине V при стоимости всех дуг c (V_A, V_B) = 1. Функции отличаются лишь вторым слагаемым, и можно показать, что значение второй функции всегда (т.е. для всех состояний), больше значения первой функции:

$$Est1(V) \le Est2(V),$$

что равнозначно

$$k(V) \le s(V).$$

Действительно, во второй функции вклад каждой фишки в общую оценку-сумму $s(V)$ либо равен 0 (фишка стоит уже на «своем» месте), либо не меньше 1 (в противном случае), в первой же функции этот вклад в $k(V)$ соответственно либо равен 0, либо равен 1.

Из последнего неравенства следует, что условие допустимости достаточно доказать только для второй функции *Est2*. Справедливость нужного условия $s(V) \le h^*(V)$ следует из следующего соображения. Если бы фишки не мешали друг другу и могли двигаться до «своего» места по кратчайшему пути, как если бы других фишек на квадрате не было, то сумма длин таких путей для всех фишек была бы в точности равна значению $s(V)$. На самом же деле фишки часто не могут двигаться по кратчайшей траектории из-за того, что на ней расположены другие фишки, поэтому длина (стоимость) оптимального решения $h^*(V)$ будет не меньше $s(V)$.

Заметим, что функция $s(V)$ не учитывает должным образом трудность обмена местами двух соседних фишек, а поэтому ее эвристическая сила в принципе может быть повышена. В ряде случаев эвристическая сила некоторой оценочной функции может быть повышена просто путем умножения на положительную константу, большую единицы, однако иногда такое повышение приводит к потере допустимости алгоритма. Например, если для игры в восемь в качестве второй составляющей эвристической функции взять $h(V) = 2 \cdot s(V)$, то в ряде случаев такая функция будет убыстрять поиск и позволит решать более трудные задачи, но условие допустимости перестанет выполняться (так как для начального состояния на рис.15: $h^*(V) \le 2 \cdot s(V)$).

Если неравенство $h_1(V) \le h_2(V)$ верно для всех вершин пространства состояний, не являющихся целевыми, A^*-алгоритм, использующий эвристическую составляющую $h_2(V)$, называется *более информированным*, чем A^*-алгоритм с функцией $h_1(V)$. Показано [Нильсон73, раздел 3.9], что если эти

функции статичны (т.е. не изменяются в процессе поиска), то более информированный алгоритм раскрывает всегда меньшее число вершин, прежде чем находит путь минимальной стоимости. Это значит, что более информированный алгоритм осуществляет более целенаправленный, а значит, более эффективный поиск целевой вершины. Таким образом, понятие информированности отражает один из аспектов понятия эвристической силы оценочной функции при поиске в пространстве состояний.

Итак, желательно подбирать такую эвристическую функцию $h(V)$, которая была бы нижней границей $h^*(V)$ (чтобы гарантировать допустимость алгоритма) и которая была бы как можно ближе к $h^*(V)$ (чтобы обеспечить эффективность поиска). Заметим, что существуют задачи, для которых нельзя найти оценочную функцию, обеспечивающую во всех случаях как эффективность, так и допустимость эвристического поиска. Поэтому часто приходится использовать эвристические функции, сокращающие поиск во многих случаях, но не гарантирующие нахождение оптимального решающего пути.

В идеальном случае, когда известна сама оценка $h^*(V)$, и она используется в качестве $h(V)$, A*-алгоритм находит оптимальный решающий путь сразу, без раскрытия ненужных вершин.

2.3.3 Алгоритм «подъема на холм»

Сильным упрощением базового алгоритма эвристического поиска с произвольной оценочной функцией является алгоритм ***«подъема на холм»*** [Нильсон85, раздел 1.1.4]. Этот алгоритм при раскрытии каждой вершины производит упорядочение (по значению оценочной функции) порожденных дочерних вершин, и выбирает для последующего раскрытия дочернюю вершину с наименьшей оценкой, а не вершину с наименьшей оценкой среди всех нераскрытых вершин дерева поиска, как в базовом алгоритме эвристического поиска. Очевидно, что такой локальный выбор среди только что построенных дочерних вершин реализовать гораздо проще, чем глобальный выбор вершины во всем дереве перебора. Реализация этого алгоритма на языке Плэнер приводится в следующем разделе настоящей главы.

Идея этого алгоритма аналогична идее известного вне области искусственного интеллекта метода «подъема на гору», применяемого для поиска максимума (или минимума) функции. Согласно этому методу, для того чтобы, в конечном счете, найти максимум функции, на каждом шаге метода производится движение в направлении наибольшей крутизны функции. Для определенного класса функций (имеющих единственный максимум и некоторые другие свойства роста) такое использование локальной информации, т.е. знания направления наиболее крутого подъема в текущей точке, позволяет найти глобальное решение, т.е. максимум функции.

В алгоритме «подъема на холм», применяемом для поиска в пространстве состояний, роль функции метода «подъема на гору» играет эвристическая оценочная функция, взятая с обратным знаком. Поиск продолжается всегда от

той дочерней вершины, которая имеет меньшее значение эвристической функции (при этом случай, когда вершин с одинаковой минимальной оценкой несколько, является нежелательным).

Важно, что алгоритм «подъема на холм» дает тот же результат, что и базовый алгоритм эвристического поиска в тех случаях, когда оценочная функция обладает определенными свойствами, в частности, имеет один (глобальный) экстремум. Алгоритм становится несостоятельным, если у эвристической функции имеется несколько локальных экстремумов. Бывают и другие случаи бесперспективности «подъема на холм»: если поверхность — множество значений функции имеет равнинный участок (как горное плато) или же участки узкого и длинного возвышения (в виде горного хребта), и процесс поиска вывел как раз на них. Таким образом, рассматриваемый алгоритм имеет ограниченную применимость, но иногда возникающие проблемы можно разрешить, построив более подходящую эвристическую функцию.

2.4 Лисп- и Плэнер-алгоритмы поиска в пространстве состояний

2.4.1 Слепой поиск на Лиспе

Реализация алгоритмов слепого перебора на языке программирования Лисп не представляет особых затруднений. Ниже приводятся тексты лисп-функций *BREADTH_FIRST_SEARCH* (поиска вширь) и *LIMITED_DEPTH_SEARCH* (ограниченного поиска вглубь) от начального состояния *StartState*. Обе функции вырабатывают в качестве своего значения либо решающий путь (в виде лисповского списка) — если таковой существует, и список () в противном случае.

Точные определения использованных в них вспомогательных функций *OPENING*, *SOLUTION*, *IS_GOAL*, *CONNECT* зависят от конкретной поисковой задачи, поэтому дадим только их общую характеристику.

Функция *OPENING* порождает для своего единственного аргумента — описания состояния задачи — список дочерних вершин-состояний (он может оказаться и пустым). Описание каждого состояния задается списком, первый элемент которого — это уникальный идентификатор состояния (число или атом), а второй элемент — собственно описание состояния задачи. Идентификаторы состояний необходимы для построения указателей функцией *CONNECT*.

Функция *CONNECT* с двумя аргументами — списком порожденных дочерних вершин и текущей (только что раскрытой, родительской) вершиной — генерирует список указателей от каждой дочерней вершины к исходной родительской.

Функция *SOLUTION* с двумя аргументами — найденным целевым состоянием и списком всех указателей дерева перебора — вырабатывает список-решение задачи (решающий путь).

Предикат *IS_GOAL* проверяет, является ли ее аргумент целевым состоянием. В случае положительного исхода проверки значением функции является само проверяемое состояние, в ином случае значение равно ().

В обоих алгоритмах поиска применяется также вспомогательная рекурсивная функция *RETAIN_NEW*, которая оставляет в списке дочерних состояний *Dlist* только те, которые не порождались ранее — тем самым исключается зацикливание при поиске в произвольном графе:

```
(defun RETAIN_NEW (Dlist)
  (prog (D)
    (cond ((null Dlist) (return ()) ))
    (setq D (car Dlist))
    (cond ((or (member D Open) (member D Closed))
             (return (RETAIN_NEW (cdr Dlist))) ))
    (return (cons D (RETAIN-NEW (cdr Dlist)))) ))
```

Другая вспомогательная рекурсивная функция *CHECK_GOALS* проверяет, есть ли среди дочерних состояний целевые. Значением этой функции является целевое состояние, если таковое найдено, и пустой список в ином случае:

```
(defun CHECK_GOALS (Dlist)
   (cond ((null Dlist) ())
         ((IS_GOAL (car Dlist)) (car Dlist))
         (t (CHECK_GOALS (cdr Dlist))) ))
```

Приведем теперь текст лисп-функции поиска вширь. В этой функции поиска (как и в следующей) идентификаторы *Open*, *Closed*, *Current* имеют тот же смысл, что и в рассмотренных в разделах 2.2.1 и 2.2.2 описаниях базовых алгоритмов. Отметим, что в отличие от базового алгоритма поиска вширь описываемая функция исключает повторно порожденные состояния (функция *RETAIN_NEW*), шаги 4 и 5 алгоритма объединены для упрощения программы. Комментариями предваряются основные шаги алгоритма.

```
(defun BREADTH_FIRST_SEARCH (StartState)
  (prog (Open Closed Current
          Deslist ;список дочерних вершин;
          Reflist ;список указателей;
          Goal ;целевая вершина;)
 ;Шаг 1:; (setq Open (list (list 'S0 StartState)))
 ;Шаг 2:; BS (cond ((null Open) (return ())))
 ;Шаг 3:; (setq Current (car Open))
          (setq Open (cdr Open))
          (setq Closed (cons Current Closed))
 ;Шаги 4,5:; (setq Deslist (OPENING Current))
          (cond((setq Goal(CHECK_GOALS Deslist))
                    (return (SOLUTION Goal Reflist)) ))
 ; Исключение повторных вершин-состояний:;
          (setq Deslist (RETAIN_NEW Deslist))
          (cond ((null Deslist) (go BS)))
          (setq Open (append Open Deslist))
 ; Построение указателей и занесение их в общий список:;
```

```
          (setq Reflist
               (append (CONNECT Deslist Current) Reflist))
          (go BS) ))
```

Нижеследующая Лисп-функция *LIMITED_DEPTH_SEARCH* поиска вглубь имеет два аргумента: кроме начального состояния *StartState* в число ее аргументов включена граничная глубина *LimDepth*. Предполагается, что в каждом списке-описании состояния, кроме идентификатора состояния и собственно описания состояния задачи, содержится также третий элемент — глубина этого состояния-вершины в дереве поиска. Проверка на окончание алгоритма (шаг 6 базового алгоритма), для упрощения программы производится в середине цикла.

```
(defun LIMITED_DEPTH_SEARCH (StartState LimDepth)
  (prog (Open Closed Current
         Deslist ;список дочерних вершин;
         Reflist ;список указателей;
         Depth   ;глубина текущей вершины; )
; Шаг 1: занесение начальной вершины (ее глубина равна 0):;
; в список нераскрытых вершин:;
    (setq Open (list (list 'S0 StartState 0)))
; Шаг 2:;
LS (cond ((null Open) (return ())))
; Шаг 3:; (setq Current (car Open))
                (setq Open (cdr Open))
                (setq Closed (cons Current Closed))
 ;Определение глубины текущей вершины:;
                (setq Depth (caddr Current))
                (cond ((IS_GOAL Current) ;/Шаг 6/;
              (return (SOLUTION Current Reflist))))
 ;Шаг 4:;       (cond ((eql Depth LimDepth) (go LS)))
 ;Шаг 5:Раскрытие вершины, исключение повторных вершин и;
 ; модификация списков указателей и нераскрытых вершин:;
                (setq Deslist (OPENING Current))
                (setq Deslist (RETAIN_NEW Deslist))
                (cond ((null Deslist) (go LS)))
   (setq Open (append (ADD_DEPTH (add1 Depth) Deslist) Open))
   (setq Reflist (append (CONNECT Deslist Current) Reflist))
                (go LS) ))
```

Функция *LIMITED_DEPTH_SEARCH* применяет все описанные выше вспомогательные функции, а также описанную ниже функцию *ADD_DEPTH*, заносящую число *Dn* (величину глубины) в описания всех дочерних состояний из списка *Dlist*:

```
(defun ADD_DEPTH (Dn Dlist)
  (cond ((null Dlist) ())
        (t (cons (list (caar Dlist) (cadar Dlist) Dn)
                 (ADD_DEPTH Dn (cdr Dlist))) )))
```

2.4.2 Поиск вглубь на Плэнере

Для сравнения трудоемкости программирования поиска вглубь на языках Лисп и Плэнер приведем текст плэнерской функции *DEPTH_FIRST_SEARCH*, выполняющей поиск вглубь от заданного состояния *St*, но без ограничения глубины (чтобы ограничить глубину, необходимо добавить в эту функцию дополнительный аргумент — граничную глубину, и несколько простых действий). Функция использует встроенный бектрекинг языка Плэнер.

Значением этой функции является список всех состояний-вершин, лежащих на решающем пути от *St* до целевой вершины. Как и ранее, вспомогательные функции *OPENING* и *IS_GOAL* зависят от конкретной решаемой задачи: первая реализует раскрытие заданной вершины-аргумента и выдает как свое значение список дочерних вершин; вторая является предикатом, проверяющим, является ли исследуемая вершина-аргумент целевой.

```
[define DEPTH_FIRST_SEARCH (lambda (St)
   [prog (Dst  %очередная раскрываемая вершина;
          Reslist %список, в котором строится решающий путь; )
         [set Dst .St] [set Reslist (.St)]
         DS [set Dst [among [OPENING .Dst]]]
         [cond ([memb .Dst .Reslist] [fail])]
         [set Reslist (!.Reslist .Dst)]
         [cond ([IS_GOAL .Dst] [return .Reslist])
                (t [go DS])] ])]
```

Из программных средств управления бектрекингом (в языке Плэнер он называется *режимом возвратов* — подробнее об этом см. раздел 4.2) в приведенном тексте используется только функция *among*, организующая точку выбора альтернатив (*развилку*). В соответствующей строке программы с меткой *DS*, являющейся началом внутреннего цикла, раскрывается очередная вершина, из получившихся дочерних вершин образуется развилка (точка бектрекинга) и выбирается одна из вершин развилки. Затем проверяется, не содержится ли выбранная вершина-состояние в списке *Reslist*, и если это так, то она добавляется в этот список, где последовательно формируется решающий путь. Далее проверяется, не является ли выбранная вершина целевой, и в зависимости от результата этой проверки либо поиск успешно заканчивается, либо производится переход на начало цикла — для того, чтобы, раскрывая текущую вершину, идти вглубь дерева поиска (каждый шаг цикла соответствует шагу вниз в дереве поиска).

В принципе алгоритм поиска вглубь мог быть запрограммирован без цикла, с использованием рекурсии. В этом случае развилка бы ставилась при каждом обращении к рекурсивной функции поиска, и в типичном случае каждое рекурсивное обращение соответствовало бы шагу вниз по дереву перебора.

Отметим, что плэнерская функция *fail*, инициирующая автоматический процесс возврата, в рассмотренной функции *DEPTH_FIRST_SEARCH* применяется только для предотвращения зацикливания, когда порожденная дочерняя вершина уже содержится в списке *Reslist*. В этом случае происходит возврат к текущей развилке для выбора следующей дочерней вершины. Возврат же к предыдущей по времени возникновения развилке (т.е. точке бектрекинга, организованной на предыдущем шаге цикла и лежащей на один шаг вверх в дереве перебора) будет инициирован самой функцией *among* в двух следующих случаях. Во-первых, когда список порожденных дочерних вершин оказался пуст, а во-вторых, когда уже перепробованы все установленные в развилке варианты (дочерние вершины) и все они оказались неуспешными.

Поскольку глубина перебора в функции *DEPTH_FIRST_SEARCH* никак не ограничена, то окончание работы алгоритма гарантировано только в случае поиска в ограниченных пространствах состояний. Если просматриваемое пространство не содержит целевого состояния, результатом этой функции будет неуспех, и процесс возврата вернет управление к последней по времени развилке, возникшей перед входом в функцию *DEPTH_FIRST_SEARCH*.

Если сравнивать эту Плэнер-функцию с вышеприведенными Лисп-функциями для поиска вширь и вглубь, то нетрудно заметить, что встроенный механизм бектрекинга позволил обойтись без многих вспомогательных структур – списков открытых и закрытых вершин, списка указателей между вершинами и др., а также без операций по их модификации. Не используются также вспомогательные функции *SOLUTION*, *CONNECT*.

Кроме достоинств, заключающихся в упрощении программирования слепого перебора вглубь, встроенный бектрекинг имеет и недостатки, главные из которых – «жесткость» и неэффективность. Встроенный механизм возвратов языка Плэнер совсем не облегчает программирование других разновидностей поиска – перебора вширь и эвристического поиска, и оно выполняется обычно без использования этого механизма, т.е. почти так же, как и в языках программирования без встроенного бектрекинга. Что же касается языка Пролог, в котором по существу нельзя отказаться от встроенного бектрекинга, то по этой причине запрограммировать на нем алгоритм поиска вширь, как и алгоритм эвристического поиска, даже сложнее, чем в обычных языках без встроенного бектрекинга.

2.4.3 Эвристический поиск на Лиспе

Возвращаясь вновь к языку Лисп, приведем текст лисповской функции *HEURISTIC_SEARCH*, реализующей эвристический поиск в пространстве состояний с использованием эвристической оценочной функции *EST*, зависящей от конкретной поисковой задачи. Вычисляемая для каждого состояния эвристическая оценка хранится как четвертый элемент списка-описания состояния. Напомним, что, как и прежде, первый элемент этого списка – уникальный идентификатор состояния, второй — собственно описание

состояния, а третий элемент — глубина соответствующей этому состоянию вершины в дереве поиска. Глубина часто используется при подсчете эвристической оценки, поэтому будем считать, что аргументом функции *EST* является список из двух элементов: собственно описания состояния и глубины соответствующей вершины.

Отметим, что на шаге 3, выбирая из списка *Open* первый элемент-вершину, мы тем самым выбираем вершину с минимальной оценкой, поскольку список *Open* всегда упорядочен по неубыванию хранящихся в нем оценок вершин-состояний (это обеспечивается вспомогательной функцией *MERGE*).

```
(defun HEURISTIC_SEARCH(StartState)
  (prog (Open Closed Current
          Deslist ;список дочерних вершин;
          Reflist ;список указателей;
          Depth   ;глубина текущей вершины;)
 ;Шаг 1:;
    (setq Open (list(list 'S0 StartState 0
 (EST (list StartState 0)) )))
 ;Шаг 2:;
 HS (cond ((null Open) (return())))
 ;Шаг 3:;
                (setq Current (car Open))
                (setq Open (cdr Open))
                (setq Closed (cons Current Closed))
                (setq Depth (caddr Current))
 ;Шаг 4:;
                (cond ((IS_GOAL Current)
               (return (SOLUTION Current Reflist))))
 ;Шаг 5:;
                (setq Deslist (OPENING Current))
 ; Исключение повторных вершин-состояний:;
                (setq Deslist (RETAIN_NEW Deslist))
                (cond ((null Deslist) (go HS)))
 ;Шаг 6:;
                (setq Open (MERGE (ADD_DEPTH_EST (add1 Depth)
Deslist) Open))
    (setq Reflist (append (CONNECT Deslist Current) Reflist))
                (go HS) ))
```

Данная лисповская функция использует (как и ранее рассмотренные функции *BREADTH_FIRST_SEARCH* и *LIMITED_DEPTH_SEARCH*) вспомогательные функции *OPENING*, *SOLUTION*, *IS_GOAL*, *CONNECT*, *EST* (которые зависят от конкретной поисковой задачи) и ранее определенную функцию *RETAIN_NEW*, а также еще две вспомогательные функции — *ADD_DEPTH_EST* и *MERGE*. Первая функция устанавливает глубину дочерних вершин и вычисляет их эвристическую оценку. Вторая функция выполняет слияние двух упорядоченных (по невозрастанию эвристической оценки) списков состояний в результирующий упорядоченный список:

```
(defun ADD_DEPTH_EST (Dn Slist)
  (cond ((null Slist) ())
         (t (cons (list (caar Slist) (cadar Slist) Dn
```

```
                        (EST (list (cadar Slist) Dn)) )
                   (ADD_DEPTH_EST Dn (cdr Slist))) ) ))

(defun MERGE (L1 L2)
  (cond ((null L1) L2)
        ((null L2) L1)
        ((> (car (cdddar L1)) (car(cdddar L2)))
           (cons (car L2) (MERGE L1 (cdr L2))))
        (t (cons (car L1) (MERGE (cdr L1) L2))) ))
```

2.4.4 Алгоритм «подъема на холм» на Плэнере

Метод «подъема на холм» можно рассматривать как обобщение поиска в глубину, при котором на каждом шаге вглубь от некоторой вершины среди ее дочерних вершин для раскрытия выбирается наиболее перспективная вершина, т.е. с наименьшей оценкой (в классическом переборе в глубину этот выбор делается произвольно). Поэтому алгоритм «подъема на холм» легко программируется на основе встроенного механизма бектрекинга языка Плэнер:

```
[define CLIMB_ON_HILL (lambda(St)
   [prog (Dst  %очередная раскрываемая вершина;
          Reslist %список вершин решающего пути; )
    [set Dst .St] [set Reslist (.St)]
 CS [set Dst [among [Q_SORT [OPENING .Dst]]]]
    [cond ([memb .Dst .Reslist] [fail])]
    [set Reslist (!.Reslist .Dst)]
    [cond ([IS_GOAL .Dst] [return .Reslist])
          (t [go CS])] ])]
```

Эта функция является усложнением ранее рассмотренной плэнерской функции *DEPTH_FIRST_SEARCH* (добавлено обращение к функции *Q_SORT*). Как и прежде, поиск ведется от начального состояния *St*, и значением функции является список всех состояний-вершин, лежащих на решающем пути от вершины *St* до цели. Вспомогательные функции *OPENING* и *IS_GOAL* зависят от конкретной решаемой задачи, а вспомогательная функция *Q_SORT* выполняет упорядочение (по неубыванию их оценок) порожденных дочерних вершин (*Q_SORT* реализует алгоритм быстрой сортировки). В ходе рекурсивного процесса сортировки применяется вспомогательная функция *DO_PARTS*, которая разбивает свой аргумент-список вершин на два списка вершин (соответственно с меньшими и большими оценками) и использует для этого разбиения эвристическую оценочную функцию *EST*, воплощающую эвристическую информацию о решаемой задаче.

```
[define Q_SORT (List)
  [prog (El Parts)
        [cond ([fin El List] [return ()] )]
        [set Parts [DO_PARTS .El .List]]
        (<Q_SORT [1 .Parts]> .El <Q_SORT [2 .Parts]>) ]]

 [define DO_PARTS (El List)
```

```
        [prog ( E (P1 ()) Key (P2 ()) )
              [set Key [EST .El]]
           D [cond ([fin E List] [return (.P1 .P2)]
                ([gt .Key [EST .E]] [set P1 (.E !.P1)])
                (t [set P2 (.E !.P2)] )]
               [go D] ]]
```

2.5 Решения модельных задач

Рассмотрим теперь реализацию вспомогательных функций *OPENING*, *SOLUTION*, *IS_GOAL*, *CONNECT* для примеров представлений задач в пространстве состояний, рассмотренных в первой главе.

2.5.1 Задача о коммивояжере (Лисп и Плэнер)

В задаче о коммивояжере состояние задачи описывается как список городов, которые коммивояжер уже посетил к настоящему моменту, и пространство состояний представляет собой дерево, причем ограниченной глубины. Поэтому равно подходящими являются все алгоритмы слепого поиска – перебор вширь и вглубь (даже без ограничения глубины). Кроме того, само найденное целевое состояние может считаться решением, поскольку представляет список городов, входящих в искомый маршрут.

Поэтому в случае программирования этой задачи на языке Лисп можно упростить соответствующие лисп-функции слепого поиска, например, *BREADTH_FIRST_SEARCH*, обойдясь без указателей и идентификаторов вершин, а также без функций *SOLUTION*, *CONNECT* и *RETAIN_NEW*. При этом потребуется убрать из текста функции строку с обращением к функции *CONNECT* и строку с обращением к функции RETAIN_NEW, а также заменить обращение к функции *SOLUTION* (аргумент встроенной функции *return*) на переменную *State*. Функции *IS_GOAL* и *OPENING* можно определить так:

```
(defun IS_GOAL (State) (eq (car(last State) 'A)))

 (defun OPENING (State)
   (prog (Dlist ;список формируемых дочерних состояний;
          TownL TownN; последний и следующий город маршрута;
          New  Roads ; списки дорог между городами; )
    (setq Roads '((A B)(A D)(A E)(A G)(B D)(B C)(C D)(C H)
                  (D E)(D H)(E G)(H G)(E H)))
 ; расширение списка путей(дорог) между городами:
                (setq Roads (ADD_WAYS Roads))
 ; последний город маршрута:  (setq TownL (car(last State)))
 ; цикл поиска соседних с городом TownL городов, в которые
 ; может вести маршрут:
                OPC (setq New (FIND TownL Roads))
                       (cond ((null New) (return Dlist)))
                       (setq TownN (car New)) (setq Roads (cadr New))
                (cond ((or (and (eq TownN 'A)(eql (length State)7))
                (not (member TownN State)) )
  (setq Dlist (cons(append State(list TownN))
```

```
Dlist))))
                        (go OPC) ))
```

Так как наличие дороги (пути) между городами *X* и *Y* означает также наличие пути между *Y* и *X*, то первоначальный список дорог между городами должен быть расширен симметричными связями-путями – это выполняет вспомогательная функция *ADD_WAYS*:

```
(defun ADD_WAYS (Rlist)
   (cond ((null Rlist) ())
         (t (cons (car Rlist)
                  (cons (list (cadar Rlist) (caar Rlist))
                        (ADD_WAYS(cdr Rlist))) ))))
```

Вспомогательная функция *FIND* выполняет поиск города-соседа для заданного города *Town*:

```
(defun FIND (Town Roads)
        (cond ((null Roads) ())
                ((eq Town (caar Roads))
               (list (cadar Roads)(cdr Roads)) )
                (t (FIND Town (cdr Roads))) ))
```

Приведем теперь для сравнения реализацию функций *IS_GOAL* и *OPENING* на языке Плэнер для их применения в функции *DEPTH_FIRST_SEARCH* поиска вглубь. Использование встроенной функции *is* аппарата сопоставления языка Плэнер (см. раздел 4.2 пособия) позволяет обойтись без вспомогательной функции поиска города-соседа:

```
[define IS_GOAL (State) [prog (X) [is (!*X A) .State]]]

[define OPENING (State)
 [prog ((Roads ((A B)(A D)(A E)(A G)(B D)(B C)
                (C D)(C H)(D E)(D H)(E G)(H G)(E H)))
         Town X Y Z
        (Dlist()) %список дочерних состояний; )
 % расширение списка путей (дорог) между городами:;
               [set Roads [ADD_WAYS .Roads]]
 % последний город маршрута:;    [set Town [-1 .State]]
 % цикл поиска следующего города маршрута:;
OPC  [cond ([not [is (!*X (.Town *Z) !*Y) .Roads]]
                        [return .Dlist] )]
               [cond ([or [and [eq .Z A] [eq [length .State] 7]]
                  [not [memb .Z .State]] ]
             [set Dlist ((!.State .Z) !.Dlist)] )]
               [set .Roads .Y]
               [go OPC] ]]

 [define ADD_WAYS (Rlist)
  [prog (R X Y)
        [cond ([fin R Rlist] ())
           ([is (*X *Y) .R]
```

```
                (.R (.Y .X) <ADD_WAYS .Rlist>) )] ]]
```

2.5.2 Задача об обезьяне (Плэнер)

В задаче об обезьяне и банане пространство состояний невелико, и хотя оно содержит много циклов, может быть использован поиск вглубь без ограничения глубины, осуществляемый функцией *DEPTH_FIRST_SEARCH*. Ниже приводятся нужные для этого Плэнер-функции, в них также используется мощная встроенная функция *is*.

Предполагается, что вершина-состояние описывается списком из двух элементов, первый из которых — собственно описание состояния, а второй — указание на оператор, в результате применения которого получено описываемое состояние, например: ((*Тб П Тя* 0) (*Перейти Тб*)). Таким образом, решающий путь будет содержать не только лежащие на нем вершины-состояния, но и операторы, связанные с дугами этого пути. Из-за такого усложнения описания состояния несколько сложнее становится предотвращающая зацикливание проверка в функции *DEPTH_FIRST_SEARCH*. Теперь для обнаружения в списке *Reslist* вершин, эквивалентных порожденному состоянию, вместо встроенной плэнерской функции *memb* нужна вспомогательная функция *SMEMB*.

```
[define IS_GOAL(State)
  [prog (X Y Z) [is (*X *Y *Z 1) [1 .State]] ]]

 [define OPENING (State)
  [prog(X Y Z
        (Dlist ()) % список дочерних вершин;
        Pc  % точка текущего положения обезьяны;
        P1 P2  %  две другие точки-константы пола;
 % список возможных комбинаций из 3 точек-констант:;
       (Points ((ТО (ТБ ТЯ)) (ТБ (ТО ТЯ)) (ТЯ (ТО ТБ)))) )
    [set State [1 .State]]  % выделение самого состояния;
 % проверка условий применимости оператора Схватить:;
    [cond ([is (ТБ Я ТБ 0) .State]
            [set Dlist (!.Dlist ((ТБ Я ТБ 1) Схватить))])]
 % определение текущей точки обезьяны и двух других точек:;
               [set Pc [1 .State]]
    [is (!*Y (.Pc (*P1 *P2)) !*Z) .Points]
 % проверка применимости операторов Взобраться и Передвинуть:;
    [cond ([is (*X П .X *Z) .State]
             [set Dlist (!.Dlist
                         ((.X Я .X .Z) Взобраться)
                         ((.P1 П .P1 .Z) (Передвинуть .P1))
                         ((.P2 П .P2 .Z) (Передвинуть .P2)) )] )]
 % проверка условий применимости оператора Перейти:;
    [cond ([is (*X П *Y *Z) .State]
             [set Dlist (!.Dlist
                         ((.P1 П .Y .Z) (Перейти .P1))
                         ((.P2 П .Y .Z) (Перейти .P2))) ])] ]]

[define SMEMB (St Rlist)
```

```
  [prog (X Y Z) [is (!*X (.St *Y) !*Z) .Rlist)] ]]
```

2.5.3 Игра в восемь (Лисп)

Граф-пространство состояний для головоломки-игры в восемь достаточно велик (в игре в пятнадцать фишек он на порядок больше), поэтому хотя в принципе применимы алгоритмы слепого поиска, предпочтителен все же эвристический поиск. Опишем нужные для *HEURISTIC_SEARCH* вспомогательные Лисп-функции *IS_GOAL*, *EST*, *OPENING*, *SOLUTION*, *CONNECT* для игры в восемь. В соответствии с предположениями, сделанными в предыдущем разделе, состояние задачи (конфигурация игры) представляется списком из следующих элементов:

- идентификатор состояния (используются атомы S_1, S_2, S_3, ... генерируемые встроенной лисповской функцией *gensym*);
- собственно описание состояния — список из номеров фишек, записанных последовательно по рядам квадрата;
- число-глубина состояния-вершины в дереве перебора;
- числовая эвристическая оценка состояния.

В описание состояния включается также — в качестве первого элемента списка — обозначение того оператора движения пустой клетки, который привел к данному состоянию. Этот элемент нужен, чтобы исключить тривиальные (см. раздел 2.2.1) повторы состояний при раскрытии вершин. Операторы будут обозначаться соответственно именами-атомами *right*, *left*, *up*, *down*; а «пустышка» (пустая клетка) — как #.

Например, список (S3 ('left 2 8 3 1 6 4 # 7 5) 1 6) описывает состояние S_3, полученное сдвигом «пустышки» влево и имеющее эвристическую оценку 6, соответствующая вершина находится в дереве перебора на глубине 1. Отметим, что эвристическая оценка используется только в алгоритме эвристического перебора, а глубина вершины — в алгоритмах эвристического перебора и ограниченного перебора вглубь.

Описанные ниже Лисп-функции для игры в восемь могут быть пригодны и для игры в пятнадцать, так как размер стороны игрового квадрата, равный 3, используется в них как глобальная переменная *Size*. Другая глобальная переменная *Goalstate* хранит описание целевого состояния (без идентификатора, глубины и эвристической оценки). Значение оценочной функции *EST* определяется как сумма числа фишек, стоящих не на «своих» местах, и длины пути (глубины) оцениваемого состояния-вершины в дереве поиска.

```
(defun IS_GOAL (State)
        (equal (cdadr State) Goalstate ))

 (defun EST (S)
  (prog (Len G N)
    (setq Len (cadr S)) (setq N 0)
```

```
          (setq S (cdar S))   (setq G Goalstate)
 ;одновременный просмотр списков-описаний
 ; заданного и целевого состояний:
 ES (cond ((null S) (return (+ N Len)) ))
 (cond ((neq(car S )(car G )) (setq N (add1 N))))
           (pop S) (pop G)  (go ES) ))

 (defun OPENING (State)
  (prog (Op St Dlist K I J El)
 ; выделение составных элементов описания состояния:
               (setq St (cadr State))
               (setq Op (car St)) (setq St (cdr St))
               (setq State St)    (setq K 0)
 ; поиск порядкового номера K «пустышки» в списке:
   OP (setq El (car St))
               (setq K (add1 K))
               (cond ((neq El '#) (setq St(cdr St)) (go OP)))
 ; вычисление номера ряда и номера столбца «пустышки»:
               (setq J (/ K Size))
(setq I (add1 (- J (rem J)))) (setq J (rem K Size))
 ; поочередно проверка возможности движения «пустышки»
 ; вправо/влево/вверх/вниз (за счет анализа оператора Op
 ; исключаем в дереве поиска тривиальные циклы (т.е. возврат
 ; после применения двух операторов в исходное состояние):
               (cond ((and (neq Op 'left) (< J Size))
             (ADD_STATE 'right K (add1 K)) ))
               (cond ((and (neq Op 'right) (> J 1))
             (ADD_STATE 'left K (sub1 K)) ))
               (cond ((and (neq Op 'down) (> I 1))
             (ADD_STATE 'up K (- K Size)) ))
               (cond ((and (neq Op 'up) (< I Size))
            (ADD_STATE 'down K (+ K Size)) ))
               (return Dlist) ))
```

При раскрытии использована вспомогательная функция *ADD_STATE*, добавляющая в список дочерних вершин список-описание новой вершины из двух элементов: идентификатора этой вершины (его генерирует функция gensym) и списка номеров фишек. В *ADD_STATE* применяется встроенная лисповская функция *NTH*, выбирающая из заданного списка элемент с заданным порядковым номером (отсчет элементов начинается с нуля).

```
(defun ADD_STATE (Op K1 K2)
   (push(list (gensym)
         (cons Op
           (cond((< K1 K2)
                   (EXCHANGE State '# (NTH (sub1 K2) State)))
                 (t (EXCHANGE State (NTH (sub1 K2) State)'#))) ))
         Dlist))
```

Рекурсивная функция *EXCHANGE*, использованная в *ADD_STATE*, формирует новое состояние игры путем перестановки заданных элементов исходного состояния-списка *List*.

```
(defun EXCHANGE (List Elem1 Elem2)
   (cond((eql Elem1 (car List))
```

```
            (cons Elem2 (EXCHANGE (cdr List) Elem1 Elem2)))
        ((eql Elem2 (car List)) (cons Elem1 (cdr List)))
    (t (cons(car List)(EXCHANGE (cdr List) Elem1 Elem2)))))
```

Функция *CONNECT* формирует список указателей от текущей вершины-состояния *Curr* к заданным (в списке *Dlist*) дочерним вершинам-состояниям. Каждый указатель есть трехэлементный список из идентификатора родительской вершины, идентификатора дочерней вершины и названия связывающего их оператора.

```
(defun CONNECT (Dlist Curr)
        (prog (D Di Ci Rlist)
  C (setq D (car Dlist)) (setq Dlist (cdr Dlist))
    (setq Di (car D))    (setq Ci (car Curr))
    (setq Rlist (cons (list Ci Di (caadr D)) Rlist))
    (cond ((null Dlist) (return Rlist)))
    (go C) ))
```

Функция *SOLUTION*, выделяя решающий путь, строит последовательность (названий) операторов, преобразующих начальное состояние в целевое (ее аргумент *Reflist* – список всех указателей-связей между состояниями). Для поиска указателя к очередной вершине решающего пути функция использует вспомогательную функцию *LOOK_FOR*.

```
 (defun SOLUTION (Goal Reflist)
   (prog (Sollist ;список, в котором строится решение;
         Gi Edge)
    (setq Gi (car Goal))
  S (cond((eq Gi 'S0) (return Sollist)))
    (setq Edge (LOOK_FOR Gi Reflist))
    (setq Sollist (cons (caddr Edge) Sollist))
    (setq Gi (car Edge))
    (go S) ))

 (defun LOOK_FOR (Id List)
   (cond ((null List) (quote error))
         ((eq ID (cadar List)) (car List))
         (t (LOOK_FOR Id (cdr List))) ))
```

В заключение приведем функцию, инициализирующую эвристический поиск решения игры в восемь: в самом его начале устанавливаются переменные-параметры встроенной функции *gensym*.

```
(defun main ()
(prog(Goalstate Initstate Size)
    (setq *gensym-prefix* 'S)
    (setq *gensym-count* 1)
    (setq Size 3)
    (setq Goalstate '(1 2 3 8 # 4 7 6 5))
    (setq Initstate '(? 2 8 3 1 6 4 7 # 5))
    (HEURISTIC_SEARCH Initstate) ))
```

2.6 Поиск на И/ИЛИ-графах

2.6.1 Особенности поиска

И/ИЛИ-граф, называемый также ***графом целей*** (задач), неявно задается при формализации задачи с использованием метода редукции, путем описания исходной задачи, операторов сведения задачи к подзадачам и множества элементарных задач.

Очевидно, что если И-вершин в этом графе нет, то получаем обычный граф-пространство состояний, для поиска в котором можно использовать рассмотренные в настоящей главе алгоритмы, но из-за наличия в общем случае в графе таких вершин нужна модификация алгоритмов, существенно их усложняющая.

Причины усложнения алгоритмов поиска на И/ИЛИ-графах связаны с тем, что более сложной становится проверка условий окончания перебора, т.е. разрешимости исходной задачи. После построения каждой новой вершины графа надо как-то определять, не стала ли разрешима начальная вершина. Если она уже разрешима, должен быть выявлен решающий граф задачи. Если же разрешимость начальной вершины все еще не установлена, то необходимо определить, имеет ли смысл продолжать поиск дальше, поскольку уже может быть ясна неразрешимость некоторых необходимых для решения исходной задачи подзадач, что означает неразрешимость самой исходной задачи. *Неразрешимость* вершины (и соответствующей задачи) дерева поиска может быть определена рекурсивным образом аналогично определению разрешимой вершины, данному в первой главе пособия:

- вершина, не являющаяся заключительной и не имеющая дочерних вершин, неразрешима;
- ИЛИ-вершина неразрешима тогда и только тогда, когда неразрешимы все ее дочерние вершины;
- И-вершина неразрешима тогда и только тогда, неразрешима хотя бы одна из ее дочерних вершин.

Как и в случае поиска в пространстве состояний, все алгоритмы перебора на И/ИЛИ-графе можно описать, опираясь на операцию раскрытия очередной вершины графа, означающую применение к ней (точнее, к соответствующему состоянию) всех возможных операторов редукции, порождающих дочерние вершины. Процесс поиска, представленный как процесс постепенного раскрытия вершин И/ИЛИ-графа, соответствует преобразованию в явную форму некоторой части всего И/ИЛИ-графа, неявно заданного при формализации задачи. Это преобразование включает в себя, вообще говоря, проведение указателей от дочерних вершин к их родительским — эти указатели необходимы для определения разрешимых и неразрешимых вершин и решающего графа (последний состоит только из разрешимых вершин).

Формируемая при поиске структура из вершин и указателей называется *графом поиска*, по окончании поиска в нем содержится решающий граф. В общем

случае граф поиска содержит разрешимые и неразрешимые вершины, а также вершины, относительно которых неизвестно – разрешимы они или нет.

Тем самым в алгоритмах поиска на И/ИЛИ-графах нужна более сложная организация списков открытых и закрытых вершин (вершины могут быть двух типов, и при этом для вершины каждого типа может быть уже установлена ее разрешимость или неразрешимость). Кроме того, в ходе поиска целесообразно тем или иным способом сохранять подграфы, являющиеся кандидатами на решающий граф задачи – по окончании поиска один из них станет решением задачи.

Основные виды перебора на И/ИЛИ-графах различаются порядком раскрытия вершин. Так же как в пространстве состояний, применяются:

- поиск вширь (вершины раскрываются в том порядке, в котором строились);
- поиск вглубь (в первую очередь раскрываются те вершины, которые были построены последними, глубина поиска обычно ограничивается);
- «подъем на холм» и эвристический поиск (выбор вершины для раскрытия производится на основе эвристических оценок вершин).

Алгоритмы поиска формулируются проще для И/ИЛИ-графов, являющихся деревьями: при этом организация перебора значительно упрощается, поскольку любая задача (цель) может встретиться в таком графе-дереве ровно один раз. Глубина вершин в формируемом алгоритмами И/ИЛИ-дереве перебора может быть определена точно так же, как в случае деревьев перебора в пространстве состояний.

Алгоритм перебора вширь на И/ИЛИ-дереве непременно обнаруживает то решающее дерево, самая глубокая заключительная вершина которого имеет минимальную глубину (если, конечно, решение задачи вообще существует).

Алгоритм неограниченного перебора вглубь может быть применен (без зацикливания) только к конечным деревьям. Более безопасным является поиск вглубь с ограничением глубины, при котором незаключительные вершины с глубиной, равной граничной, просто считаются неразрешимыми. Как и при переборе в пространстве состояний, ограничение глубины может воспрепятствовать нахождению решения, но существующее в пределах ограничения решающее дерево будет найдено обязательно.

Слепой поиск на И/ИЛИ-дереве, как и в пространстве состояний, неэффективен, поэтому предпочтителен алгоритм эвристического поиска, при котором для очередного раскрытия выбирается наиболее перспективная вершина в дереве перебора, точнее, перспективная вершина в поддереве, являющемся кандидатом на решающее дерево.

Для эвристического поиска в И/ИЛИ-графе был предложен и исследован алгоритм, являющийся аналогом A*-алгоритма в пространстве состояний, и получивший название ***AO*-алгоритма*** [Нильсон73, разделы 5.6, 5.7; Нильсон85, раздел 3.2]. Если задать стоимости дуг в И/ИЛИ-дереве и применять для эвристического поиска оценочную функцию специального вида, то этот алгоритм найдет за конечное число шагов решающее дерево с минимальной

стоимостью. В простейшем случае стоимость дерева может пониматься как суммарная стоимость входящих в него дуг. Иными словами, для АО*-алгоритма верна аналогичная теорема о допустимости, гарантирующая нахождение решения с минимальной стоимостью (так называемого *оптимального дерева*) при условии, что дерево решения существует.

2.6.2 Схема просмотра И/ИЛИ-дерева

Отвлекаясь от многочисленных деталей организации поиска на И/ИЛИ-графах общего вида, рассмотрим только на схему раскрытия вершин и формирования решающего дерева при поиске в графах-деревьях. Процесс просмотра И/ИЛИ-дерева проще всего организовать на основе трех взаимно рекурсивных процедур, назовем их соответственно *AND/OR, AND и OR-процедурой* и опишем основные их шаги.

AND/OR-процедура, получая на вход некоторую вершину-задачу *Goal*, вырабатывает либо неуспех, если эта вершина неразрешима, либо возвращает решающее дерево, показывающее ее разрешимость и являющееся решением этой задачи.

1. **Шаг 1.** Проверить, является ли заданная вершина *Goal* заключительной. Если да, то закончить процедуру, выдав саму эту вершину в качестве результата, в ином случае перейти к следующему шагу.
2. **Шаг 2.** Проверить, можно ли редуцировать (раскрыть) заданную вершину *Goal*. Если да, то переход на шаг 3, в противном случае выход из процедуры с сообщением о неуспехе.
3. **Шаг 3.** Раскрыть заданную вершину *Goal*, сформировав список ее дочерних вершин *Glist*, переход на шаг 4.
4. **Шаг 4.** Если вершина *Goal* является И-вершиной, то применить к ней и списку *Glist* AND-процедуру, после чего закончить текущую процедуру с полученным (AND-процедурой) результатом.
5. **Шаг 5.** Если вершина *Goal* является ИЛИ-вершиной, то применить к ней и списку *Glist* OR-процедуру, и завершить текущую процедуру с полученным (OR-процедурой) результатом.
6. Конец AND/OR-процедуры.

Две следующие процедуры похожи друг на друга. Каждая из них получает на вход вершину *Goal* соответствующего типа и список ее дочерних вершин *Glist*, результат процедуры — либо решающее дерево с корнем в вершине *Goal*, либо сообщение о неуспехе (неразрешимости этой вершины).

Основные шаги AND-процедуры:

1. **Шаг 1.** Установить в качестве решающего графа для задачи *Goal* саму эту вершину и перейти на следующий шаг.
2. **Шаг 2.** Если список *Glist* не пуст, то перейти на шаг 3, в ином случае закончить текущую процедуру, выдав в качестве результата сформированный решающий граф для задачи-цели *Goal*.

3. **Шаг 3.** Выбрать из списка *Glist* очередную вершину (назовем ее *Current*) и проверить ее разрешимость, обратившись к AND/OR-процедуре.
4. **Шаг 4.** Если *Current* неразрешима, то выход из текущей процедуры с сообщением о неуспехе, иначе перейти на следующий шаг.
5. **Шаг 5.** Подсоединить полученный AND/OR-процедурой решающий граф для вершины *Current* к решающему графу вершины *Goal* и перейти на шаг 2.
6. Конец AND-процедуры.

Основные шаги OR-процедуры:

1. **Шаг 1.** Установить в качестве решающего графа для задачи *Goal* саму эту вершину и перейти на следующий шаг.
2. **Шаг 2.** Если список *Glist* не пуст, то перейти на шаг 3, в ином случае выход из текущей процедуры с сообщением о неуспехе.
3. **Шаг 3.** Выбрать из списка *Glist* очередную вершину (назовем ее *Current*) и проверить ее разрешимость, обратившись к AND/OR-процедуре.
4. **Шаг 4.** Если *Current* неразрешима, то переход на шаг 2, в противном случае – на следующий шаг.
5. **Шаг 5.** Включить полученный AND/OR-процедурой решающий граф для вершины *Current* в решающий граф вершины *Goal* и закончить текущую процедуру, выдав полученный решающий граф в качестве ее результата.
6. Конец OR-процедуры.

Рассмотренные рекурсивные процедуры могут быть легко детализированы для получения либо простого поиска вглубь (при произвольном выборе вершины на шаге 3 OR-процедуры) или получения алгоритма «подъема на холм» (если список *Glist* упорядочен по неубыванию эвристической оценочной функции и на шаге 3 OR-процедуры выбирается вершина с наименьшей оценкой). Для реализации же эвристического поиска потребуется другая схема просмотра И/ИЛИ-дерева — по причине необходимости доступа к концевым вершинам поддерева, являющегося кандидатом на решающее дерево.

2.6.3 Различия и ключевые операторы

В заключение целесообразно обсудить ряд общих идей, на которых может быть основана редукция задач. К их числу, прежде всего, относится идея выделения так называемых ключевых операторов и различий состояний. Будем далее предполагать, что задачи и подзадачи решаются в пространстве состояний и записываются как (S_I, O, S_G), где S_I и S_G — соответственно начальное и целевое состояния, причем множество O операторов преобразований состояний может быть опущено в случае его постоянности (неизменности).

Часто при поиске в пространстве состояний нетрудно обнаружить один оператор, который обязательно должен входить в решение задачи (по сути, применение этого оператора есть необходимый шаг решения этой задачи).

В графе-пространстве состояний этому оператору обычно соответствует дуга, связывающая две практически отдельные части графа. Такой оператор и называется ***ключевым оператором*** в пространстве состояний. К примеру, в задаче о пирамидке ключевым оператором был оператор переноса на нужный колышек нижнего диска 3 (см. рис.8).

Ключевой оператор может быть использован для следующего способа сведения исходной задачи к подзадачам. Пусть

- Op — найденный в пространстве состояний задачи ключевой оператор ($Op \in O$);
- S_{Op} — множество состояний, к которым применим ключевой оператор Op;
- $Op(s)$ — состояние, полученное в результате применения ключевого оператора Op к состоянию s ($s \in S_{Op}$).

Тогда исходную задачу можно свести к трем подзадачам:

- первая задача (S_I, S_{Op}) состоит в поиске пути от начального состояния исходной задачи к одному из состояний, к которому применим ключевой оператор Op;
- вторая задача является элементарной, она заключается в применении этого ключевого оператора;
- третья задача ($Op(s)$, S_G) состоит в поиске пути от состояния, полученного применением ключевого оператора, к целевому состоянию исходной задачи.

Важен порядок решения перечисленных задач: третья задача не может быть решена раньше первой и второй, так же как вторая – раньше первой. Для решения первой и третьей задач опять может быть применен метод редукции, или же их решение может быть найдено непосредственным поиском в пространстве состояний.

Для большинства задач не удается всегда однозначно выделить ключевой оператор, гораздо чаще удается найти множество *операторов-кандидатов* в ключевые, т.е. операторов, с большой вероятностью могущих стать ключевыми. Таким образом, в общем случае необходим процесс перебора операторов-кандидатов, каждый из которых образует свое множество результирующих задач (этот перебор и означает поиск на И/ИЛИ-графе задачи).

При таком способе редукции задач самый важный вопрос состоит в том, как найти кандидаты в ключевые операторы. Один из способов, предложенных и опробованных впервые в одной из наиболее известных систем искусственного интеллекта GPS [Слейгл, глава 8], заключается в выявлении ***различий*** для начального и целевого состояний задачи. Различие легче всего формализовать как несоответствие различных элементов описаний начального и целевого состояний. Если начальное состояние является целевым, то различий между ними нет, и задача решена.

К примеру, в задаче об обезьяне и банане различием можно считать неравенство соответствующих элементов списков, описывающих два состояния. Тогда при сравнении состояний ($Т_О$, $П$, $Т_Я$, 0) и ($Т_Я$, $П$, $Т_О$, 1)

выявляются три различия — соответственно в первых, третьих и четвертых элементах этих списков.

С каждым различием в системе GPS был связан один или несколько операторов, призванных устранять или уменьшать это различие. Эти операторы и являлись по сути кандидатами в ключевые. На каждом этапе работы система определяла различие между текущим состоянием (объектом) задачи и целевым состоянием (объектом), а затем выбирала и пыталась применить оператор для уменьшения найденного различия. Операторы могли включать *предусловия* (условия применимости), выполнение которых было необходимо для их успешного применения, в этом случае GPS сводила исходную задачу к задаче достижения нужного условия.

В задаче об обезьяне естественно связать различия и операторы-кандидаты в ключевые следующим образом:

- Различие в первом элементе списка-описания состояния (положение обезьяны в плоскости пола) — операторы *Перейти* и *Передвинуть*.
- Различие во втором элементе (положение обезьяны по вертикали) — оператор *Взобраться*.
- Различие в третьем элементе (положение ящика) — оператор *Передвинуть*.
- Различие в четвертом элементе (содержимое руки обезьяны) — оператор *Схватить*.

Ясно, что для реализации рассмотренных идей в виде алгоритма или программы нужна, во-первых, специальная процедура сравнения описаний состояний и вычисления различий. Во-вторых, нужна таблица или процедура, связывающая ключевые операторы с возможными различиями (в GPS использовалась таблица связей различие-оператор). Последняя процедура (или таблица) должна также устанавливать приоритет различий и операторов, т.е. порядок их выбора в случаях, когда выявлено несколько различий и возможно применение нескольких ключевых операторов. Различия должны быть упорядочены по степени их значимости для решения исходной задачи. Система GPS начинала с попытки обработки более серьезных и трудно устранимых различий, переходя затем к более легким.

В задаче об обезьяне и банане приоритет (значимость) различий можно установить, например, следующим образом: различие в четвертом, затем во втором, затем в третьем и первом элементах описания состояния задачи. Такой же приоритет может быть и у операторов, уменьшающих эти различия.

Подчеркнем, что воплощаемая в указанных процедурах информация является специфической, зависящей от конкретной задачи, т.е. эвристической проблемно-ориентированной информацией. Именно с этим связана одна из слабостей применяемого в системе GPS подхода: процедуры вычисления различий и уменьшающих их операторов должны быть отдельно реализованы для каждой конкретной задачи (или для очень узкой предметной области, включающей несколько видов задач), в противном случае снижалась эффективность решения задач.

В то же время основной механизм решения задач в системе GPS не был проблемно-ориентированным: он представлял собой реализацию универсального эвристического метода решения задач, часто применяемого человеком, и известного как *анализ целей и средств* (*means-ends analysis*). Ключевая идея этой эвристики такова:

- поиск различий между тем, что дано в поставленной задаче, и тем, что надо получить;
- последовательное устранение найденных различий с помощью подходящих средств–операций.

Работая в соответствии с этой эвристикой, GPS применяла несколько схем редукции задач, и на основе выявления различий между объектами задачи и применения уменьшающих эти различия операторов рекурсивно формировала систему (дерево) целей (задач и подзадач).

ГЛАВА 3. ПОИСК НА ИГРОВЫХ ДЕРЕВЬЯХ

3.1 Деревья игры. Поиск выигрышной стратегии

Будем рассматривать класс ***игр двух лиц с полной информацией***. В таких играх участвуют два игрока, которые поочередно делают свои ходы. В любой момент каждому игроку известно все, что произошло в игре к этому моменту и что может быть сделано в настоящий момент. Игра заканчивается либо выигрышем одного игрока (и проигрышем другого), либо ничьей.

Таким образом, в рассматриваемый класс не попадают игры, исход которых зависит хотя бы частично от случая — большинство карточных игр, игральные кости, «морской бой» и проч. Тем не менее класс достаточно широк: в него входят такие игры, как шахматы, шашки, реверси, калах, крестики-нолики и другие игры.

Для формализации и изучения игровых стратегий в классе игр с полной информацией может быть использован подход, основанный на редукции задач. Напомним, что при этом должны быть определены следующие составляющие: форма описания задач и подзадач; операторы, сводящие задачи к подзадачам; элементарные задачи; а также задано описание исходной задачи.

Наиболее интересной представляется задача поиска выигрышной стратегии для одного из игроков, отправляясь от некоторой конкретной позиции игры (не обязательно начальной). При использовании подхода, основанного на редукции задач, выигрышная стратегия ищется в процессе доказательства того, что игра может быть выиграна. Аналогично, поиск ничейной стратегии, исходя из некоторой конкретной позиции, ведется в процессе доказательства того, что игра может быть сведена к ничьей.

Ясно, что описание решаемой задачи должно содержать описание конфигурации игры, для которой ищется нужная стратегия. Например, в шашках игровая позиция включает задание положений на доске всех шашек, в том числе дамок. Обычно описание конфигурации содержит также указание, кому принадлежит следующий ход.

Пусть именами игроков будут ПЛЮС и МИНУС. Будем использовать следующие обозначения:

- X^S или Y^S – некоторая конфигурация игры, причем индекс S принимает значения + или −, указывая, кому принадлежит следующий ход (т.е. в конфигурации X^+ следующий ход должен делать игрок ПЛЮС, а в X^- игрок МИНУС);
- $W(X^S)$ — задача доказательства того, что игрок ПЛЮС может выиграть, исходя из конфигурации X^S;
- $V(X^S)$ — задача доказательства того, что игрок МИНУС может выиграть, отправляясь от конфигурации X^S.

Рассмотрим сначала игровую задачу $W(X^S)$. Операторы сведения (редукции) этой задачи к подзадачам определяются исходя из ходов, допустимых в проводимой игре:

- Если в некоторой конфигурации X^+ очередной ход делает игрок ПЛЮС, и имеется N допустимых ходов, приводящих соответственно к конфигурациям X_1^-, X_2^-, ..., X_N^-, то для решения задачи $W(X^+)$ необходимо решить по крайней мере одну из подзадач $W(X_i^-)$. Действительно, так как ход выбирает игрок ПЛЮС, то он выиграет игру, если хотя бы один из ходов ведет к выигрышу — см. рис.16(а).
- Если же в некоторой конфигурации Y^- ход должен сделать игрок МИНУС и имеется K допустимых ходов, приводящих к конфигурациям Y_1^+, Y_2^+, ..., Y_K^+, то для решения задачи $W(Y^-)$ требуется решить каждую из возникающих подзадач $W(Y_i^+)$. Действительно, поскольку ход выбирает МИНУС, то ПЛЮС выиграет игру, если выигрыш гарантирован ему после любого хода противника — см. рис.16(б).

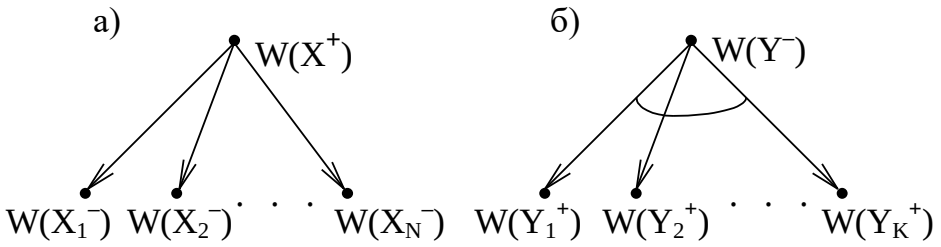

Рис. 16. Редукция игровых задач

Последовательное применение к исходной конфигурации игры данной схемы редукции порождает И/ИЛИ-дерево (И/ИЛИ-граф), которое называют ***деревом*** (***графом***) ***игры***. Дуги игрового дерева соответствуют ходам игроков, вершины — конфигурациям игры, причем листья дерева —это позиции, в которых игра завершается выигрышем, проигрышем или ничьей. Некоторые листья являются заключительными вершинами, соответствующими элементарным задачам — позициям, выигрышным для игрока ПЛЮС. Заметим, что для конфигураций, где ход принадлежит ПЛЮСу, в игровом дереве получается ИЛИ-вершина, а для позиций, в которых ходит МИНУС, получается И-вершина.

Цель построения игрового дерева или графа — получение решающего поддерева для задачи $W(X^S)$, показывающего, как игрок ПЛЮС может выиграть игру из позиции X^S независимо от ответов противника. Для этого могут быть применены разные алгоритмы поиска на И/ИЛИ-графах. Решающее дерево заканчивается на позициях, выигрышных для ПЛЮСа, и содержит полную стратегию достижения им выигрыша: для каждого возможного продолжения игры, выбранного противником, в дереве есть ответный ход, приводящий к победе.

Для задачи $V(X^S)$ схема сведения игровых задач к подзадачам аналогична: ходам игрока ПЛЮС будут соответствовать И-вершины, а ходам МИНУСа —

ИЛИ-вершины, заключительные же вершины будут соответствовать позициям, выигрышным для игрока МИНУС.

Конечно, подобная редукция задач применима и в случае, когда нужно доказать существование ничейной стратегии в игре. При этом определение элементарной задачи должно быть соответствующим образом изменено.

Рассмотрим в качестве иллюстрации простую игру, называемую «последний проигрывает». Два игрока поочередно берут по одной или две монеты из кучки, первоначально содержащей семь монет. Игрок, забирающий последнюю монету, проигрывает.

На рис.17 показан полный граф игры для задачи $V(7^+)$, жирными дугами на нем выделен решающий И/ИЛИ-граф, который доказывает, что второй игрок (т.е. игрок МИНУС, начинающий вторым), всегда может выиграть. Конфигурация игры описана как число оставшихся в текущий момент монет, также указание, кому принадлежит следующий ход. Заключительные вершины, соответствующие элементарной задаче $V(1^+)$, т.е. выигрышу игрока МИНУС, в графе подчеркнуты.

Представленная в решающем графе выигрышная стратегия может быть сформулирована словесно так: если в очередном ходе игрок ПЛЮС берет одну монету, то в следующем ходе МИНУС должен взять две, а если ПЛЮС берет две монеты, то МИНУС должен забрать одну. Отметим, что для аналогичной задачи $W(7^+)$ решающий граф построить не удастся (начальная вершина неразрешима); таким образом, у игрока ПЛЮС нет выигрышной стратегии в этой игре.

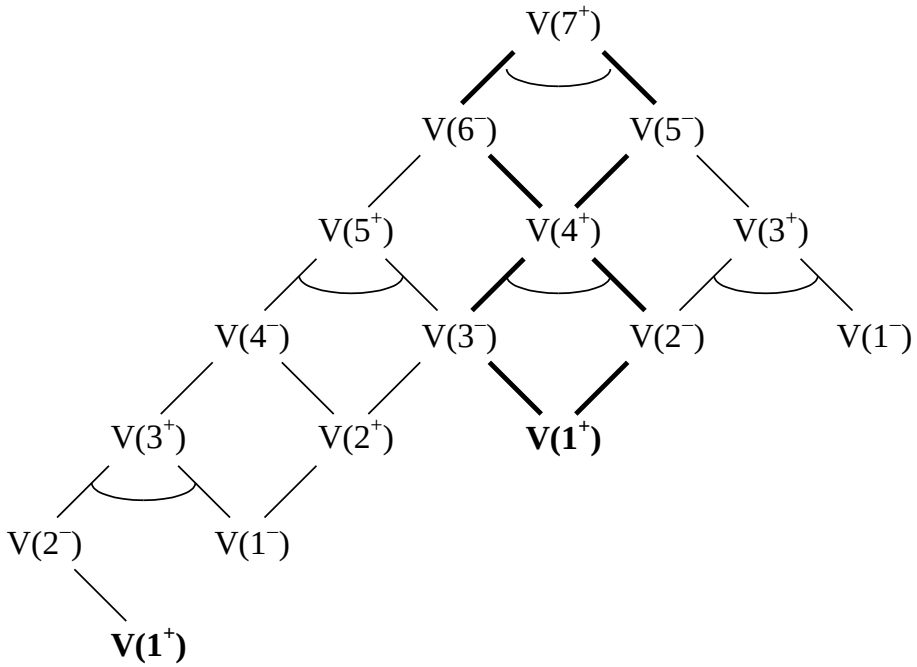

Рис. 17. Граф игры «последний проигрывает»

В большинстве игр, представляющих интерес, таких как шашки и шахматы, построить полные решающие деревья и найти полные игровые стратегии не представляется возможным. Например, для шашек число вершин

в полном игровом дереве оценивается величиной порядка 10^{40}, и просмотреть такое дерево практически нереально. Алгоритмы же упорядоченного перебора с применением эвристик не настолько уменьшают просматриваемую часть дерева игры, чтобы дать существенное, на несколько порядков, сокращение времени поиска.

Тем не менее для неполных игр в шашки и шахматы (например, для эндшпилей), как и для всех несложных игр, таких как «крестики-нолики» на квадрате небольшого размера, можно успешно применять алгоритмы поиска на И/ИЛИ-графах, позволяющие обнаруживать выигрышные и ничейные игровые стратегии.

Рассмотрим, к примеру, игру «крестики-нолики» на квадрате 3×3. Игрок ПЛЮС ходит первым и ставит крестики, а МИНУС — нолики. Игра заканчивается, когда составлена либо строка, либо столбец, либо диагональ из крестиков (в этом случае выигрывает ПЛЮС) или ноликов (выигрывает МИНУС). Оценим размер полного дерева игры: начальная вершина имеет 9 дочерних вершин, каждая из которых в свою очередь имеет 8 дочерних; каждая вершина глубины 2 имеет 7 дочерних и т.д. Таким образом, число концевых вершин в дереве игры равно 9! = 362 880, но многие пути в этом дереве обрываются раньше на заключительных вершинах. Значит, в этой игре возможен полный просмотр дерева и нахождение выигрышной стратегии. Однако ситуация изменится при существенном увеличении размеров квадрата или же в случае неограниченного поля игры.

В таких случаях, как и во всех сложных играх, вместо нереальной задачи поиска полной игровой стратегии решается, как правило, более простая задача — поиск для заданной позиции игры *достаточно хорошего* первого хода.

3.2 Минимаксная процедура

С целью поиска достаточно хорошего первого хода просматривается обычно часть игрового дерева, построенного от заданной конфигурации. Для этого применяется один из переборных алгоритмов (вглубь, вширь или эвристический) и некоторое искусственное окончание перебора вершин в игровом дереве: например, ограничивается время перебора или же глубина поиска.

В построенном таким образом частичном дереве игры оцениваются его концевые вершины (листья), и по полученным оценкам определяется наилучший ход от заданной игровой конфигурации. При этом для оценивания концевых вершин полученного дерева используется так называемая ***статическая оценочная функция***, а для получения оценки остальных вершин — корневой (начальной) и промежуточных между корневой и концевыми — используется так называемый ***минимаксный принцип***.

Статическая оценочная функция, будучи применена к некоторой вершине-конфигурации игры, дает числовое значение, оценивающее

различные достоинства этой игровой позиции. Например, для шашек могут учитываться такие (статические) элементы конфигурации игры, как продвинутость и подвижность шашек, количество дамок, контроль ими центра и проч. По сути, статическая функция вычисляет эвристическую оценку шансов на выигрыш одного из игроков. Для определенности будем рассматривать задачу выигрыша игрока ПЛЮС и соответственно поиска достаточно хорошего его первого хода от заданной конфигурации.

Будем придерживаться общепринятого соглашения, по которому значение статической оценочной функции тем больше, чем больше преимуществ имеет игрок ПЛЮС (над игроком МИНУС) в оцениваемой позиции. Очень часто оценочная функция выбирается следующим образом:

- статическая оценочная функция положительна в игровых конфигурациях, где игрок ПЛЮС имеет преимущества;
- статическая оценочная функция отрицательна в конфигурациях, где МИНУС имеет преимущества;
- статическая оценочная функция близка к нулю в позициях, не дающих преимущества ни одному из игроков.

Например, для шашек в качестве простейшей статической функции может быть взят перевес в количестве шашек (и дамок) у игрока ПЛЮС. Для игры «крестики-нолики» на фиксированном квадрате возможна такая статическая оценочная функция:

$$E(P) = \begin{cases} +\infty, & \text{если P есть позиция выигрыша игрока ПЛЮС} \\ -\infty, & \text{если P есть позиция выигрыша игрока МИНУС} \\ (N_L^+ + N_C^+ + N_D^+) - (N_L^- + N_C^- + N_D^-) & \text{в остальных случаях} \end{cases}$$

где:

- $+\infty$ — очень большое положительное число;
- $-\infty$ — очень маленькое отрицательное число;
- N_L^+, N_C^+, N_D^+ — соответственно число строк, столбцов и диагоналей, «открытых» для игрока *ПЛЮС* (т. е. где он еще может поставить три выигрышных крестика подряд);
- N_L^-, N_C^-, N_D^- — аналогичные числа для игрока *МИНУС*.

На рис.18 приведены две игровые позиции (на квадрате 4×4) и соответствующие значения статической оценочной функции.

а)

	O		
	X	X	

E (P) = 8 - 5 = 3

б)

		O	
		X	
		X	
	O	X	

E (P) = 6 - 4 = 2

Рис. 18. Примеры статических оценок позиций игры "крестики-нолики"

Подчеркнем, что с помощью статической оценочной функции оцениваются только концевые вершины дерева игры, для оценок же промежуточных вершин, как и начальной вершины, используется минимаксный принцип, основанный на следующей простой идее. Если бы игроку ПЛЮС пришлось бы выбирать один из нескольких возможных ходов, то он выбрал бы наиболее сильный ход, т.е. ход, приводящий к позиции с наибольшей оценкой. Аналогично, если бы выбирать ход пришлось игроку МИНУС, то он выбрал бы ход, приводящий к позиции с наименьшей оценкой.

Сформулируем теперь сам минимаксный принцип:

- ИЛИ-вершине дерева игры приписывается оценка, равная максимуму оценок ее дочерних вершин;
- И-вершине игрового дерева приписывается оценка, равная минимуму оценок ее дочерних вершин.

Минимаксный принцип положен в основу ***минимаксной процедуры***, предназначенной для определения наилучшего (более точно, достаточно хорошего) хода игрока исходя из заданной конфигурации игры *S* при фиксированной глубине поиска *N* в игровом дереве. Предполагается, что игрок ПЛЮС ходит первым (т.е. начальная вершина есть ИЛИ-вершина). Основные этапы этой процедуры таковы:

1. Дерево игры строится (просматривается) одним из известных алгоритмов перебора (как правило, алгоритмом поиска вглубь) от исходной позиции *S* до глубины *N*.
2. Все концевые вершины полученного дерева, т.е. вершины, находящиеся на глубине *N*, оцениваются с помощью статической оценочной функции.
3. В соответствии с минимаксным принципом вычисляются оценки всех остальных вершин: сначала вычисляются оценки вершин, родительских для концевых, затем — родительских для этих родительских вершин и т.д.; такое оценивание вершин продолжается при движении снизу вверх по дереву поиска до тех пор, пока не будут оценены вершины, дочерние для начальной вершины, т.е. для исходной конфигурации *S*.
4. Среди вершин, дочерних к начальной, выбирается вершина с наибольшей оценкой: ход, который к ней ведет, и есть искомый наилучший ход в игровой конфигурации *S*.

На рис.19 показано применение минимаксной процедуры для дерева игры, построенного до глубины $N = 3$. Концевые вершины не имеют имен, они обозначены своими оценками — значениями статической оценочной функции. Числовые индексы имен остальных вершин показывают порядок, в котором эти вершины строились алгоритмом перебора вглубь. Рядом с этими вершинами находятся их минимаксные оценки, полученные при движении в обратном (по отношению к построению дерева) направлении. Таким образом, наилучший ход — первый из двух возможных.

На данном игровом дереве выделена ветвь (последовательность ходов игроков), представляющая так называемую *минимаксно-оптимальную* игру, при которой каждый из игроков всегда выбирает наилучший для себя ход. Заметим, что оценки всех вершин этой ветви дерева совпадают, и оценка начальной вершины равна оценке концевой вершины этой ветви.

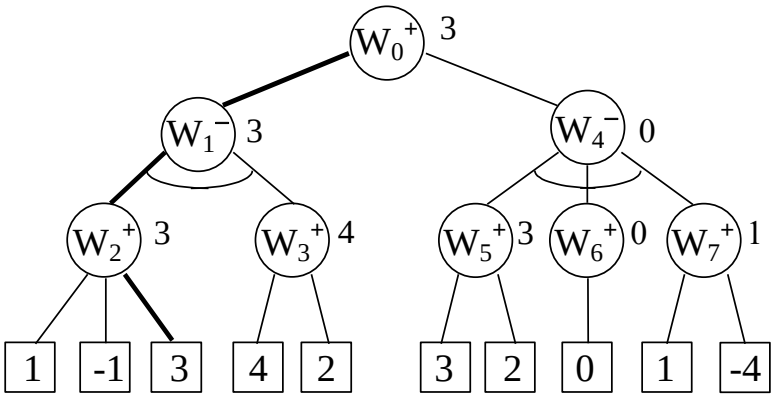

Рис. 19. Игровое дерево, построенное минимаксной процедурой

В принципе статическую оценочную функцию можно было бы применить и к промежуточным вершинам, и на основе этих оценок осуществить выбор наилучшего первого хода, например, сразу выбрать ход, максимизирующий значение статической оценочной функции среди вершин, дочерних к исходной. Однако считается, что оценки, полученные с помощью минимаксной процедуры, более надежные меры относительного достоинства промежуточных вершин, чем оценки, полученные прямым применением статической оценочной функции. Действительно, минимаксные оценки основаны на просмотре игры вперед и учитывают разные особенности, которые могут возникнуть в последующем, в то время как простое применение оценочной функции учитывает лишь свойства позиции как таковой. Это отличие статических и минимаксных оценок существенно для «активных», динамичных позиций игры (например, в шашках и шахматах к ним относятся конфигурации, в которых возникает угроза взятия одной или нескольких фигур). В случае же так называемых «пассивных» (спокойных) позиций статическая оценка может мало отличаться от оценки по минимаксному принципу.

3.3 Альфа-бета процедура

В минимаксной процедуре процесс построения частичного дерева игры отделен от процесса оценивания вершин, и это приводит к тому, что в целом минимаксная процедура оказывается неэффективной стратегией поиска хорошего первого хода. Для повышения эффективности поиска необходимо вычислять оценки (статические и минимаксные) вершин одновременно с построением игрового дерева и не рассматривать вершины, которые хуже уже построенных вершин. Это приводит к так называемой ***альфа-бета процедуре*** поиска наилучшего первого хода от заданной позиции, в основе которой лежит

достаточно очевидное соображение для сокращения поиска: если есть два варианта хода одного игрока, то худший в ряде случаев можно сразу отбросить, не выясняя, насколько в точности он хуже.

Рассмотрим сначала идею работы альфа-бета процедуры на примере игрового дерева, приведенного на рис.19. Дерево игры строится до глубины $N = 3$ алгоритмом перебора вглубь. Причем сразу, как это становится возможным, вычисляются не только статические оценки концевых вершин, но и *предварительные* минимаксные оценки промежуточных вершин. Предварительная оценка определяется, соответственно, как минимум или максимум уже известных к настоящему моменту оценок дочерних вершин. В общем случае, эта оценка может быть получена при наличии оценки хотя бы одной дочерней вершины. В ходе дальнейшего построения дерева игры и получения новых оценок вершин предварительные оценки постепенно уточняются, опять же по минимаксному принципу.

Пусть таким образом построены вершины W_1^-, W_2^+ и первые три конечные вершины (листья) — см. рис.20. Эти листья оценены статической функцией, и вершина W_2^+ получила точную минимаксную оценку 3, а вершина W_1^- — предварительную оценку 3. Далее при построении и раскрытии вершины W_3^+ статическая оценка первой ее дочерней вершины дает величину 4, которая становится предварительной оценкой самой вершины W_3^+. Эта предварительная оценка будет потом, после построения второй ее дочерней вершины, пересчитана. Причем согласно минимаксному принципу оценка может только увеличиться (поскольку подсчитывается как максимум оценок дочерних вершин), но даже если она увеличится, это не повлияет на оценку вершины W_1^-, поскольку последняя при уточнении по минимаксному принципу может только уменьшаться (так как равна минимуму оценок дочерних вершин). Следовательно, можно пренебречь второй дочерней вершиной для W_3^+, не строить и не оценивать ее, поскольку уточнение оценки вершины W_3^+ не повлияет на оценку вершины W_1^-. Такое сокращение поиска в игровом дереве называется ***отсечением*** ветвей.

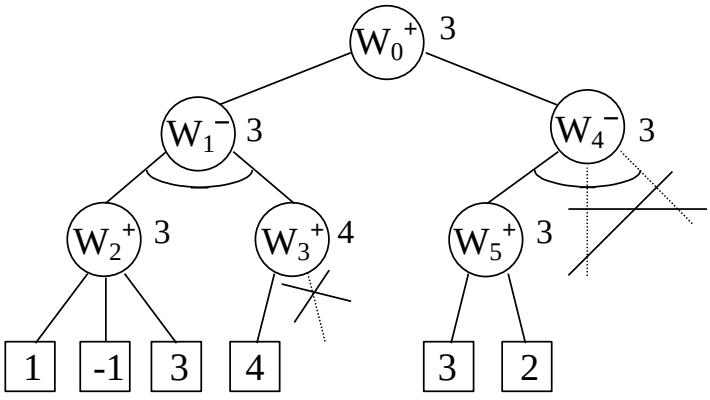

Рис. 20. Игровое дерево, построенное альфа-бета процедурой

Продолжим для нашего примера процесс поиска в глубину одновременным вычислением предварительных (и точных, где это

возможно) оценок вершин вплоть до момента, когда построены уже вершины W_4^-, W_5^+ и две дочерних последней, которые оцениваются статической функцией. Исходя из оценки первой дочерней вершины начальная вершина W_0^+, соответствующая исходной позиции игры, к этому моменту уже предварительно оценена величиной 3. Вершина W_5^+ получила точную минимаксную оценку 3, а ее родительская W_4^- получила пока только предварительную оценку *3*. Эта предварительная оценка вершины W_4^- может быть уточнена в дальнейшем, но в соответствии с минимаксным принципом возможно только ее уменьшение, и это уменьшение не повлияет на оценку вершины W_0^+, поскольку последняя, опять же согласно минимаксному принципу, может только увеличиваться. Таким образом, построение дерева можно прервать ниже вершины W_4^-, отсекая целиком выходящие из нее вторую и третью ветви (и оставляя ее оценку предварительной).

На этом построение игрового дерева заканчивается, полученный результат — лучший первый ход — тот же самый, что и при минимаксной процедуре. У некоторых вершин дерева осталась неуточненная, предварительная оценка, однако этих приближенных оценок оказалось достаточно для того, чтобы определить точную минимаксную оценку начальной вершины и наилучший первый ход. В то же время произошло существенное сокращение поиска: вместо 17 вершин построено только 11 и вместо 10 обращений к статической оценочной функции понадобилось всего 6.

Обобщим рассмотренную идею сокращения перебора. Сформулируем сначала правила вычисления оценок вершин дерева игры, в том числе предварительных оценок промежуточных вершин, которые для удобства будем называть ***альфа-*** и ***бета-величинами***:

- концевая вершина игрового дерева оценивается статической оценочной функцией сразу, как только она построена;
- промежуточная вершина предварительно оценивается по минимаксному принципу, как только стала известна оценка хотя бы одной из ее дочерних вершин; каждая предварительная оценка пересчитывается (уточняется) всякий раз, когда получена оценка еще одной дочерней вершины;
- предварительная оценка ИЛИ-вершины (альфа-величина) полагается равной наибольшей из вычисленных к текущему моменту оценок ее дочерних вершин;
- предварительная оценка И-вершины (бета-величина) полагается равной наименьшей из вычисленных к текущему моменту оценок ее дочерних вершин.

Укажем очевидное следствие этих правил вычисления: альфа-величины не могут уменьшаться, а бета-величины не могут увеличиваться.

Сформулируем теперь правила прерывания перебора, или отсечения ветвей игрового дерева:

А. Перебор можно прервать ниже любой И-вершины, бета-величина которой не больше, чем альфа-величина одной из предшествующих ей ИЛИ-вершин (включая корневую вершину дерева).

Б. Перебор можно прервать ниже любой ИЛИ-вершины, альфа-величина которой не меньше, чем бета-величина одной из предшествующих ей И-вершин.

В случае А говорят, что имеет место ***альфа-отсечение***, поскольку отсекаются ветви дерева, начиная с ИЛИ-вершин, которым приписана альфа-величина, а в случае В — ***бета-отсечение***, поскольку отсекаются ветви, начинающиеся с бета-величин.

Важно, что рассмотренные правила работают в ходе построения игрового дерева вглубь; это означает, что предварительные оценки промежуточных вершин появляются лишь по мере продвижения от концевых вершин дерева вверх к корню и реально отсечения могут начаться только после того, как получена хотя бы одна точная минимаксная оценка промежуточной вершины.

В рассмотренном на рис.20 примере первое прерывание перебора было бета-отсечением, а второе — альфа-отсечением. Причем в обоих случаях отсечение было неглубоким, поскольку необходимая для соблюдения соответствующего правила отсечения альфа- или бета-величина находилась в непосредственно предшествующей к точке отсечения вершине. В общем же случае она может находиться существенно выше отсекаемой ветви – где-то на пути от вершины, ниже которой производится отсечение, к начальной вершине дерева, включая последнюю.

После прерывания перебора предварительные оценки вершин в точках отсечения остаются неуточненными, но, как уже отмечалось, это не препятствует правильному нахождению предварительных оценок всех предшествующих вершин, как и точной оценки корневой вершины и ее дочерних вершин, а значит, и искомого наилучшего первого хода.

На приведенных выше правилах вычисления оценок вершин и выполнения отсечений (всюду, где это возможно) основана альфа-бета процедура, являющаяся более эффективной реализацией минимаксного принципа. В принципе, используя математическую индукцию и рассуждая, как показано на примере, несложно доказать следующее

Утверждение:

Альфа-бета процедура всегда приводит к тому же результату (наилучшему первому ходу), что и простая минимаксная процедура той же глубины.

Важным является вопрос о том, насколько в среднем альфа-бета процедура эффективнее обычной минимаксной процедуры. Нетрудно заметить, что количество отсечений в альфа-бета процедуре зависит от степени, в которой предварительные оценки (альфа- и бета-величины), полученные первыми, аппроксимируют окончательные минимаксные оценки: чем ближе эти оценки, тем больше отсечений и меньше перебор. Это положение иллюстрирует пример

на рис. 20, в котором ветвь дерева с минимаксно-оптимальной игрой строится практически в самом начале поиска.

Таким образом, эффективность альфа-бета процедуры зависит от порядка построения и раскрытия вершин в дереве игры. Возможен и неудачный порядок просмотра, при котором придется пройти без отсечений через все вершины дерева, и в этом, худшем, случае альфа-бета процедура не будет иметь никаких преимуществ по сравнению с минимаксной процедурой.

Наибольшее число отсечений достигается, когда при переборе в глубину первой обнаруживается конечная вершина, статическая оценка которой станет в последствии минимаксной оценкой начальной вершины. При максимальном числе отсечений строится и оценивается минимальное число концевых вершин. Показано [Нильсон73, раздел 5.14], что в случае, когда самые сильные ходы рассматриваются первыми, количество концевых вершин глубины N, которые будут построены и оценены альфа-бета процедурой, примерно равно числу концевых вершин, которые были бы построены и оценены на глубине N/2 обычной минимаксной процедурой. Таким образом, при фиксированном времени и памяти альфа-бета процедура сможет пройти при поиске вдвое глубже по сравнению с обычной минимаксной процедурой.

В заключение отметим, что статическая оценочная функция и альфа-бета процедура — две непременные составляющие почти всех компьютерных игровых программ (в том числе коммерческих). Часто используются также дополнительные эвристические приемы в самой альфа-бета процедуре [Слейгл, глава 2; Нильсон 73, раздел 5.14]:

- направленное (эвристическое) отсечение неперспективных ветвей: например, построение дерева игры обрывается на «пассивных» позициях и к ним применяется статическая оценочная функция, для «активных» же позиций поиск продолжается дальше до нужной глубины или даже глубже, поскольку на это можно использовать время, сэкономленное вследствие отсечения ветвей;
- последовательное углубление, при котором альфа-бета процедура применяется неоднократно: сначала до некоторой небольшой глубины (например, 2), а затем глубина увеличивается с каждой итерацией, причем после каждой итерации выполняется переупорядочение всех дочерних вершин — с тем, чтобы увеличить число отсечений в последующих итерациях;
- фиксированное упорядочение вершин при спуске-построении дерева вглубь, при котором в первую очередь строится и раскрывается дочерняя вершина, оцениваемая как более перспективная (эта оценка может быть проведена как с помощью статической оценочной функции, так и более простой, хотя и менее надежной эвристической функции);
- динамическое упорядочение вершин, при котором каждый раз после уточнения минимаксных оценок и проведения отсечений производится переупорядочивание вершин во всем построенном

к текущему моменту дереве (с помощью некоторой эвристической функции) и для дальнейшего раскрытия выбирается наиболее перспективная вершина (заметим, что по существу это означает переход от классического перебора вглубь к алгоритму упорядоченного перебора на И/ИЛИ-графах).

Для усиления игры могут быть также использованы библиотеки типовых игровых ситуаций [Братко, глава 15] и другие идеи [Лорьер, глава 6].

3.4 Плэнер-алгоритмы поиска на игровых деревьях

Рассмотрим сначала описание на языке Плэнер функции *MIN_MAX*, реализующей минимаксную процедуру. Аргументы этой функции: *Instate* – исходная позиция игры, для которой ищется наилучший ход; *N* – глубина поиска, т.е. количество ходов вперед. Вырабатываемое функцией значение — это дочерняя для *Instate* позиция, соответствующая искомому наилучшему ходу.

```
[define MIN_MAX (lambda (Instate N)
   [SELECT_MAX [MM_EVALP .Instate 0 T]] )]
```

Функция *MIN_MAX* использует две вспомогательные функции: *SELECT_MAX* и *MM_EVALP*. Первая функция, *SELECT_MAX* выбирает из своего единственного аргумента-списка, состоящего из позиций и их оценок, позицию с наибольшей оценкой:

```
[define SELECT_MAX (lambda (List)
  [prog (Elem  Max_elem)
      [fin Max_elem List]
   SM [cond ([fin Elem List] [return [2 .Max_elem]])
            ([gt [1 .Elem] [1 .Max_elem]]
                      [set Max_elem .Elem])]
      [go SM]] )]
```

Функция *MM_EVALP* с тремя аргументами является главной рекурсивной функцией, оценивающей вершины дерева игры по минимаксному принципу. На каждом шаге рекурсии она оценивает вершину-позицию *Position*, находящуюся на глубине *Depth* и имеющую тип *Deptype*: "ИЛИ" при *Deptype* = T и "И" при *Deptype* = (). Исходная позиция (корневая вершина дерева) имеет тип *T* и находится на глубине 0. Значением функции *MM_EVALP* является либо вычисленная оценка вершины *Position* (при *Depth* > *0*) либо список дочерних для *Position* позиций с их числовыми оценками (при *Depth* = 0).

При работе *MM_EVALP* используются вспомогательные функции, определение которых зависит от конкретной игры: *OPENING*, вычисляющая для заданной позиции игры список дочерних вершин-позиций, и *STAT_EST* — статическая оценочная функция.

```
[define MM_EVALP (lambda (Position Depth Deptype)
  [prog (D  %дочерняя позиция;
         (Movelist ())   %список ходов-позиций);
         Pvalue Dvalue)  %оценки текущей и дочерней
                          позиций;
% 1: установка развилки, включающей все дочерние вершины текущей
позиции (список () добавлен в развилку, чтобы "поймать" момент ее
закрытия);
   [set D [among (<OPENING .Position> ())]]
% 2: если развилка закрыта - возврат функцией подсчитанной оценки;
   [cond ([empty .D]
            [return [cond ([eq .Depth 0] .Movelist)
                          (t .Pvalue) ]] )]
% 3: вычисление оценки очередной дочерней позиции: либо применение
статической функции, либо рекурсивный спуск;
   [cond ([eq .Depth [- .N 1]]
            [set Dvalue [STAT_EST .D]])
         (t [set Dvalue [MM_EVALP .D [+ 1 .Depth]
                                  [not .Deptype]]] )]
% 4: пересчет оценки текущей позиции по минимаксному принципу;
   [cond ([hasval Pvalue] [pset Pvalue
              [cond (.Deptype [max .Pvalue .Dvalue])
                    (t [min .Pvalue .Dvalue]) ]])
         (t [pset Pvalue .Dvalue]) ]
% 5: при необходимости пересчет для исходной позиции списка
ходов(дочерних позиций) с их оценками;
   [cond ([eq .Depth 0]
           [pset Movelist (!.Movelist (.Dvalue .D))] )]
% 6: возврат к другой альтернативе развилки;
   [fail]  ])]
```

Теперь опишем на Плэнере функцию *ALPHA_BETA*, реализующую альфа-бета процедуру. Она имеет сходную с *MIN_MAX* структуру:

```
[define ALPHA_BETA (lambda (Instate N)
   [SELECT_MAX [AB_EVALP .Instate 0 T () () ]] )]
```

Аргументы этой функции: *Instate* — заданная позиция игры, для которой ищется лучший ход, *N* — глубина поиска. Значением функции *ALPHA_BETA* является дочерняя для *Instate* вершина-позиция, соответствующая наилучшему ходу.

Как и *MIN_MAX*, функция *ALPHA_BETA* использует две вспомогательные функции: *SELECT_MAX* и *AB_EVALP*. Первая из этих функций была определена выше, она выбирает из своего аргумента-списка позицию с наибольшей оценкой.

Вторая функция, *AB_EVALP* — главная рекурсивная функция, которая эффективно оценивает вершины дерева игры по минимаксному принципу с выполнением отсечений. Она имеет пять аргументов: *Pos*, *Depth*, *Deptype*, *Amax*, *Bmin*, и на каждом шаге рекурсии вычисляет оценку вершины-позиции *Pos*, находящейся на глубине *Depth* и имеющей тип *Deptype* ("ИЛИ" при *Deptype* = T и "И" при *Deptype* = ()). Для проведения отсечений функция

использует вспомогательные величины *Amax* и *Bmin*, являющиеся соответственно максимальной из альфа-величин и минимальной из бета-величин вершин, предшествующих оцениваемой вершине *Pos*. Значением функции *AB_EVALP* (как и *MM_EVALP*) является либо вычисленная оценка позиции *Pos* (при *Depth* > 0), либо список дочерних для *Pos* позиций с их числовыми оценками (при *Depth* = 0).

При первом обращении к функции *AB_EVALP* (внутри функции *ALPHA_BETA*) смысл ее аргументов такой: исходная позиция *Pos* (т.е. корневая вершина) имеет тип *T*, находится на глубине 0, величины *Amax* и *Bmin* неизвестны. Эти величины необходимы для проверки условий правил отсечений, поэтому отсечения не могут быть выполнены, пока величины не определены (альфа-отсечение — пока неизвестна величина *Bmin*, а бета-отсечение – пока неизвестна *Amax*). Эти значения станут известны только лишь, когда часть дерева игры будет просмотрена до заданной глубины.

Функция *AB_EVALP* (как и *MM_EVALP*) использует вспомогательные функции, определение которых зависит от конкретной игры: *OPENING*, вычисляющую для заданной позиции игры список дочерних вершин-позиций, и *STAT_EST* — статическую оценочную функцию.

```
[define AB_EVALP (lambda (Pos Depth Deptype Amax Bmin)
  [prog (D (Movelist ()) Pvalue Dvalue)
% 1: установка развилки из всех дочерних для Pos вершин (список ()
добавлен в развилку, чтобы "поймать" момент ее закрытия);
   [set D [among (<OPENING .Pos> ())]]
% 2: если развилка закрыта, то возврат функцией подсчитанной оценки;
   [cond ([empty .D]
            [return [cond ([eq .Depth 0] .Movelist)
                          (t .Pvalue)] ] )]
% 3: проверка возможности отсечения и отсечение - возврат на
предыдущий уровень рекурсии (встроенная функция permex реализует
досрочный выход с заданным значением из заданной функции, при этом
уничтожает все развилки, возникшие после входа в заданную функцию);
   [cond ([and [neq .Depth 0] [hasval Pvalue]]
           [cond (.Deptype
             [cond ([and [num .Bmin] [ge .Pvalue .Bmin]]
                     [permex .Pvalue AB_EVALP]) ])
                 (t
             [cond ([and [num .Amax] [le .Pvalue .Amax]]
                     [permex .Pvalue AB_EVALP])] )])]
% 4: вычисление оценки очередной дочерней вершины: либо применение
статической оценочной функции, либо рекурсивное обращение к
рассматриваемой функции;
   [cond ([eq .Depth [- .N 1]]
            [set Dvalue [STAT_EST .D]])
         (t [set Dvalue [AB_EVALP .D  [+ 1 .Depth]
                                [not .Deptype]
         [cond ([and [hasval Pvalue] .Deptype]
                 [cond ([num .Amax] [max .Amax .Pvalue])
                       (t .Pvalue)])
               (t .Amax)]
         [cond ([and [hasval Pvalue] [not .Deptype]]
                 [cond ([num .Bmin] [min .Bmin .Pvalue])
                       (t .Pvalue) ])
```

```
               (t .Bmin)] ]] )]
% 5: пересчет предварительной оценки текущей позиции по минимаксному
принципу;
   [cond ([hasval Pvalue] [Pset  Pvalue
                  [cond (.Deptype [max .Pvalue .Dvalue])
                        (t [min .Pvalue .Dvalue])] ])
         (t [pset Pvalue .Dvalue] )]
% 6: в случае исходной позиции пересчитываем список ходов (дочерних
позиций) с их оценками;
   [cond ([eq .Depth 0]
             [pset Movelist (!.Movelist (.Dvalue .D))] )]
% 7: возврат к другой альтернативе развилки ;
   [fail] ])]
```

Обе рассмотренные процедуры, *MIN_MAX* и *ALPHA_BETA*, должны быть соответствующим образом модифицированы, если при просмотре позиций на заданную глубину N возможно появление выигрышной или проигрышной позиции, обрывающих некоторые ветви игрового дерева. В этом случае к таким позициям необходимо сразу применить статическую оценочную функцию.

Заметим в заключении, что в случае выбора языка Лисп для реализации процедур поиска на игровых деревьях минимаксная процедура может быть запрограммировано весьма кратко и понятно, особенно при использовании функционалов. Что же касается альфа-бета процедуры, то программа на Лиспе [Уинстон, глава 13] существенно более громоздка и трудна для понимания, чем приведенная выше Плэнер-функция, поскольку в последней применяется встроенный механизм возвратов.

ГЛАВА 4. КРАТКИЕ СВЕДЕНИЯ О ЯЗЫКАХ ЛИСП И ПЛЭНЕР

4.1 Язык программирования Лисп

В языке Лисп все действия описываются в виде *функций*. С точки зрения синтаксиса, обращения к функциям, как и обрабатываемые ими данные, представляют собой так называемые *S-выражения*. В простейшем случае S-выражением является *атом* (идентификатор или число), в более сложном — *список*, который представляет собой заключенную в круглые скобки последовательность элементов списка. Лисповские списки имеют рекурсивную структуру, поскольку элементом списка может быть произвольное S-выражение — как атом, так и список. Примеры списков:

```
(()(a b 1(c))class)
(45 89 (()()))
```

Пустой список () имеет в Лиспе и другое обозначение — *nil*, причем он одновременно относится и к атомам, и к спискам.

Некоторые *S*-выражения можно вычислять, такие выражения называются *формами*. Простейшими формами являются числа и атомы-константы *T* и *nil*, а также переменные с уже определенными значениями; значением такой формы служит соответственно само число, сам атом *T* или *nil* или же текущее значение переменной.

Формой является также обращение к функции, т.е. список вида

```
(f a₁ a₂ ... aₙ)    (n≥0)
```

где *f* — имя функции, а a_i — ее аргументы, которые для *обычной функции* должны быть формами. В Лиспе есть и *специальные функции*, у которых может быть произвольное число аргументов и аргументы которых либо не вычисляются, либо вычисляются особым образом.

Программа на Лиспе представляет собой последовательность форм, и ее выполнение заключается в последовательном вычислении этих форм. Как правило, в начале программы определяются новые функции, а затем следуют обращения к ним.

Языка Лисп предоставляет большой набор *встроенных* (стандартных) *функций*, на основе которых составляется пользовательская программа. Ниже кратко описаны некоторые из встроенных функций — те, что были использованы в предыдущих разделах пособия. Все эти функции входят в наиболее известные версии Лиспа — *Common Lisp* и *MuLisp* [Семенов].

Отметим, что комментарием в языке Лисп считается любой текст между двумя точками с запятой «;».

Определение новых функций

Для этого служит встроенная функция *defun*, к которой возможно любое из следующих обращений:

```
(defun f (lambda(v1 v2 ... vn) e))        (n≥0)
(defun f (v1 v2 ... vn) e)
```

Значением этой функциональной формы является имя определяемой функции, т.е. *f*. Побочным же эффектом вычисления этой формы будет определение новой лисповской функции с именем *f*, аргументами (формальными параметрами) v_i и телом *e* — формой, зависящей от v_i.

Отметим, что таким образом определяется обычная Лисп-функция, т.е. функция с фиксированным количеством аргументов, которые всегда вычисляются при обращении к ней.

При последующем обращении в программе к новой функции:

```
(f a1 a2 ... an)
```

сначала вычисляются аргументы функции (фактические параметры) a_i, затем вводятся локальные переменные v_i, которым присваиваются значения соответствующих аргументов a_i, а далее вычисляется форма-тело *e* при этих значениях переменных v_i, после чего эти переменные уничтожаются. Вычисленное значение формы становится значением функции *f*.

Операции над списками

```
(car l)
```

Значением аргумента *l* должен быть непустой список; значением же функции является первый элемент верхнего уровня этого списка.

```
(cdr l)
```

Значением аргумента *l* должен быть непустой список, а значением функции является «хвост» этого списка, т.е. список, полученный отбрасыванием первого элемента.

Например, если переменная *X* имеет своим значением список (*p*(*q r*)), то значение формы (*car X*) равно *p*, а значение (*cdr X*) равно ((*q r*)).

Кроме этих двух функций часто используются функции, задающие их суперпозиции. Имена всех таких функций начинаются на букву *c*, а заканчиваются на букву *r*, между ними же может стоять произвольная комбинация из не более чем четырех букв *a* и *d*, причем буква a означает наличие в суперпозиции функции *car*, а буква *d* — функции *cdr*. Сама же

последовательность букв соответствует порядку следования функций *car* и *cdr* в эквивалентной суперпозиции. Например:

$$(caddr\ l) \equiv (car(cdr(cdr\ l)))$$

Предполагается, что список-аргумент *l* у всех этих функций-суперпозиций, как и у ниже описываемой функции *nth*, содержит необходимое число элементов (в противном случае вычисление программы прерывается сообщением об ошибке).

```
(nth n l)
```

Значением аргумента *n* должно быть положительное целое число (обозначим его *N*), а значением аргумента *l* – список. Значением функции является *N*-й от начала элемент этого списка, причем элементы списка нумеруются, начиная с 0.

К примеру, если переменная X имеет своим значением список (*p*(*q r*)7), то значение формы (*nth* 2 *X*) равно 7, так же как и значение формы (*caddr X*).

Все рассмотренные выше Лисп-функции называются *селекторами*, так как они выбирают определенные элементы списков. Рассмотрим теперь функции, называемые *конструкторами*: они строят новый список, который и выдают в качестве своего результата.

```
(cons e l)
```

Эта функция строит список, первым элементом которого будет значение аргумента *e*, а хвостом списка — значение аргумента *l*.

```
(append l1 l2)
```

Данная функция осуществляет слияние (конкатенацию) элементов двух списков (их верхнего уровня) в один результирующий список.

```
(list e1 e2 ... en)        (n≥1)
```

Эта функция имеет произвольное количество аргументов, из значений которых она строит список (количество элементов на верхнем уровне результирующего списка равно количеству аргументов).

```
(last l)
```

Значением функции является список из одного элемента – последнего элемента верхнего уровня списка-аргумента *l*.

Например, если переменная *X* имеет своим значением список (*p*(*q r*)), а переменная *Y* — список (*t*), то:

– значение формы (*cons X Y*) равно ((*p*(*q r*))*t*),
– значение формы (*list X Y*) равно ((*p*(*q r*))(*t*)),
– значение формы (*append X Y*) равно (*p*(*q r*) *t*);
– значение формы (*last X*) равно ((*q r*)).

Арифметические функции

```
(length l)
```

Значением аргумента *l* должен быть список. Функция вычисляет количество элементов (верхнего уровня) этого списка. К примеру, значение (*length X*) равно 2, если переменная *X* имеет значение (*p* (*q r*)).

Значениями аргументов следующих функций должны быть числа, над которыми производятся арифметические операции.

```
(add1 n)
```

Функция прибавляет число 1 к числу-аргументу и выдает результат в качестве своего значения.

```
(sub1 n)
```

Эта функция вычитает 1 из значения своего аргумента и выдает результат в качестве своего значения.

```
(+ n1 n2)
```

Значением функции является сумма значений ее аргументов.

```
(- n1 n2)
```

Значением этой функции является разность значений ее аргументов.

```
(* n1 n2)
```

Значением этой функции является произведение значений ее аргументов.

```
(/ n1 n2)
```

Результат вычисления этой функции — частное, получающееся при делении первого числа на второе.

```
(rem n1 n2)
```

Результат вычисления этой функции — остаток от деления первого числа на второе.

Предикаты

Предикатом называется форма, значение которой рассматривается как логическое значение «истина» или «ложь». Особенностью языка Лисп является то, что «ложью» считается пустой список, записываемый как () или *nil*, а «истиной» — произвольное выражение, отличное от () и *nil*. В частности, в роли «истины» может использоваться атом *T*.

```
(null e)
```

Эта функция проверяет, является ли значение ее аргумента пустым списком; если да, то значение функции равно *T*, иначе — *nil*.

```
(eq e1 e2)
```

Функция сравнивает значения своих аргументов, которые должны быть атомами-идентификаторами. В случае их равенства (идентичности) значение функции равно *T*, иначе — *nil*.

```
(eql e1 e2)
```

В отличие от предыдущей функции, данная функция сравнивает значения своих аргументов, которыми могут быть не только атомы-идентификаторы, но и атомы-числа. Если аргументы равны, то значение функции равно *T*, иначе — *nil*.

```
(equal e1 e2)
```

Функция производит сравнение двух произвольных *S*-выражений — значений своих аргументов. Если они равны (т.е. представляют собой одно и то же *S*-выражение), то значение функции равно *T*, иначе — *nil*.

```
(neq e1 e2)
```

Аналог предыдущей функции, но значения аргументов сравниваются на «не равно».

```
(member a l)
```

Значением первого аргумента должен быть атом, а второго — список. Функция производит поиск заданного атома на верхнем уровне заданного списка. В случае успеха поиска в качестве значения функции выдается хвост списка *l*, начиная с найденного атома, в случае же неуспеха выдается *nil*.

```
(= n1 n2)      (/= n1 n2)
(> n1 n2)      (>= n1 n2)
(< n1 n2)      (<= n1 n2)
```

Значениями аргументов этих функций должны быть числа. Эти числа сравниваются соответственно на равенство, неравенство, «больше», «больше или равно», «меньше», «меньше или равно», и если сравнение успешно, вырабатывается атом *T*, иначе — *nil*.

Логические функции

Так называются три функции, реализующие основные логические операции.

```
(not  e)
```

Эта функция реализует логическое отрицание и является, по сути, дубликатом функции *null*: если значение аргумента равно *nil* («ложь»), то функция выдает результат *T* («истина»), а при любом другом значении аргумента в качестве результата выдается *nil*.

Следующие функции являются особыми, так как, во-первых, они имеют произвольное число аргументов, а во-вторых, не обязательно вычисляют значения всех своих аргументов.

(and e_1 e_2 ... e_k) ($k \geqslant 1$)

Эта функция реализует конъюнкцию. Функция по очереди вычисляет свои аргументы. Если значение очередного из них равно *nil* («ложь»), то функция, не вычисляя оставшиеся аргументы, заканчивает свою работу со значением *nil*, а иначе переходит к вычислению следующего аргумента. Если функция дошла до вычисления последнего аргумента, то с его значением она и заканчивает свою работу.

(or e_1 e_2 ... e_k) ($k \geqslant 1$)

Эта функция реализует дизъюнкцию. Функция по очереди вычисляет свои аргументы. Если значение очередного из них не равно *nil* («ложь»), то функция, не вычисляя оставшиеся аргументы, заканчивает свою работу со значением этого аргумента, в противном случае она переходит к вычислению следующего

аргумента. Если функция дошла до вычисления последнего аргумента, то с его значением она и заканчивает свою работу.

К числу логических функций можно отнести и лисповское *условное выражение*:

```
(cond (p1 e1,1 e1,2 ... e1,k1) ... (pn  en,1 en,2 ...  en,kn))
      (n⩾1,  ki⩾1)
```

Функция последовательно вычисляет первые элементы своих аргументов — предикаты p_i. Если все они имеют значение *nil* («ложь»), тогда функция заканчивает свою работу с этим же значением. Но если был обнаружен предикат p_i, значение которого отлично от *nil*, тогда функция уже не будет рассматривать остальные предикаты, а последовательно вычислит все формы $e_{i,j}$ из соответствующего аргумента (ветви условного выражения) и со значением последней из них закончит свою работу.

Отметим, что за исключением последних форм в каждой ветви (они используются для вычисления значения условного выражения), в качестве форм $e_{i,j}$ имеет смысл использовать только такие, которые имеют при вычислении побочный эффект, например, присваивание переменной значения и т.п.

Другие специальные функции

```
(quote e)   или   'e
```

Это функция блокировки вычислений: она выдает в качестве значения свой аргумент, не вычисляя его. Например, значением формы *'*(*car* (2)) является само выражение (*car* (2)).

```
(gensym)
```

Это функция генерации уникальных атомов (символов): при каждом обращении к ней выдается новый атом-идентификатор. Этот идентификатор получается склеиванием специального префикса и очередного номера (целого числа). Префикс и целое число, от которого начинается нумерация генерируемых атомов, могут быть установлены заранее, как, например, в языке MuLisp:

```
(setq *gensym-prefix* 'S)   (setq *gensym-count* 2)
```

После этого при последовательных обращениях к функции *gensym* она будет выдавать атомы S_2, S_3, S_4 и т.д.

Блочная и связанные с ней функции

```
(prog (v1 v2 ... vn) e1 e2 ... ek)      (n≥0, k≥1)
```

Эту специальную функцию называют «блочной», поскольку ее вычисление напоминает выполнение блоков в других языках программирования. Вычисление функции начинается с того, что вводятся локальные переменные v_i, перечисленные в ее первом аргументе, и всем им в качестве начального значения присваивается пустой список *nil*. После этого функция последовательно вычисляет остальные свои аргументы — формы e_i. Вычислив последнюю из них, функция *prog* заканчивает работу со значением этой формы, уничтожив перед этим все свои локальные переменные v_i.

Поскольку только значение последней формы e_k используется как значение всей функции *prog*, то в качестве остальных форм e_i имеет смысл использовать только функции с побочным эффектом. Некоторые из этих функций перечислены ниже.

В качестве одной из форм e_i может быть записан атом-идентификатор, в этом случае он не вычисляется, а трактуется как метка, на которую будет производиться переход внутри этого блока (функцией *go*).

```
(return e)
```

Это функция досрочного выхода из блока. Она может использоваться только внутри блочной функции *prog*, поскольку завершает вычисление ближайшей объемлющей блочной функции, устанавливая ее значение равным значению аргумента *e*.

```
(go  e)
```

Функция реализует переход по метке. Аргумент ее не вычисляется, в качестве ее аргумента должен быть задан идентификатор — одна из меток ближайшей объемлющей блочной функции. Функция *go* полностью завершает вычисление той формы этой блочной функции, в которую она входит (на любом уровне), и осуществляет переход на вычисление формы, непосредственно следующей за указанной меткой.

```
(setq v e)
```

Это аналог оператора присваивания. В качестве аргумента *v* (он не вычисляется) должно быть задано имя переменной, существующей в данный момент. Функция присваивает этой переменной новое значение — вычисленное значение формы *e*. Это же значение является значением и самой функции *setq*, однако оно, как правило, не используется.

Следующие две особые функции используются для упрощения записи часто используемых конструкций (*setq v*(*cdr v*)) и (*setq v*(*cons*(*e v*)).

```
(pop v)
```

Аргументом этой функции (он не вычисляется) должно быть имя переменной, существующей в данный момент и имеющей своим значением непустой список. Хвост этого непустого списка присваивается в качестве нового значения указанной переменной, а также выступает в качестве значения самой функции *pop*.

```
(push e v)
```

В качестве второго аргумента этой функции (он не вычисляется) должно быть задано имя переменной, в качестве первого — произвольная форма. Функция вычисляет эту форму и строит новый список, первый элемент которого — вычисленное значение, а хвост — список, являющийся значением переменной *v*. Результирующий список становится новым значением переменной *v* и значением самой функции *push*.

Например, если переменная *X* имеет значение (*d*(*e*)*g*), а переменная *U* — значение (1 2), то значение формы (*pop X*) равно ((*e*)*g*), а значение формы (*push U X*) равно ((1 2)*d*(*e*)*g*).

4.2 Язык программирования Плэнер

В языке Плэнер программа и обрабатываемые ею данные записываются в виде *выражений*, к которым относятся: *атомы* (идентификаторы и числа), *обращения к переменным*, состоящие из префикса и имени переменной (например, *.X*), и *списки*, представляющие собой последовательности выражений заключенные в круглые, квадратные или угловые скобки (например, [*elem 1*(*a b c*)]). Отметим, что в Плэнере идентификатор *nil* не имеет никакого отношения к пустому списку.

Некоторые выражения можно вычислять, получая в результате значения (ими являются выражения); такие выражения называются *формами*. К ним относятся: атомы (значение атома - сам атом), обращения к переменным с префиксом «.» (значением является текущее значение переменной), списки в круглых скобках (значением является список из значений элементов исходного списка) и обращения к функциям, имеющие вид [$f\ a_1\ a_2\ \ldots\ a_n$], где f — имя функции, а a_i — ее аргументы. То, что форма *e* имеет значение *v*, будем обозначать следующим образом: $e \Rightarrow v$.

Программа на Плэнере представляет собой последовательность форм, ее выполнение заключается в вычислении этих форм.

В языке имеется много *встроенных* (стандартных) *функций*. Ниже вкратце описаны некоторые из них — те, что использованы в предыдущих разделах пособия.

Отметим, что комментарием в Плэнере считается любой текст между знаком процента «%» и ближайшей точкой с запятой «;».

Определение новых функций

Для определения новой функции следует обратиться к встроенной функции *define*:

```
[define f dexp]
```

Вычисление функции *define* в качестве побочного эффекта приводит к появлению в программе новой функции с именем *f* и с так называемым определяющим выражением *dexp*, которое должно иметь следующий вид:

```
(lambda (v1 v2 ... vn) e)     (n≥0)
```

где v_i — формальные параметры новой функции, а *e* — форма, зависящая от v_i .

При последующем обращении к этой новой функции

```
[f a1 a2 ... an]
```

сначала вычисляются аргументы (фактические параметры) a_i, затем вводятся локальные переменные v_i, которым присваиваются значения соответствующих аргументов a_i, и далее вычисляется форма *e* при этих значениях переменных v_i, после чего эти переменные уничтожаются. Значение данной формы становится значением функции *f* при аргументах a_i .

Операции над списками

```
elem n l]
```

Значением аргумента *n* должно быть ненулевое целое число (обозначим его *N*), а значением второго аргумента — список *L* с любыми скобками. Значением функции является *N*-й от начала элемент списка *L*, если $N > 0$, или $|N|$-й от конца элемент этого списка, если $N < 0$. В случае, когда в качестве *n* явно указано число, имя функции в обращении можно опустить: например, [1 .*X*] является сокращением для [*elem* 1 .*X*].

Если требуется вычислить список в круглых скобках

```
(e1 e2 ... en)     (n≥0)
```

т.е. если он рассматривается как форма, то все его элементы должны быть формами. Значением такого списка является список (с круглыми скобками) из значений его элементов. Например, если переменная *X* имеет значение (*a b*), то

```
(.X c [1 .X])  ⇒  ((a b) c a)
```

В таких списках можно использовать и *сегментные формы*, к которым относятся сегментные обращения к переменным (с префиксом «!.», например: *!.X*) и сегментные обращения к функциям (в угловых скобках, например: <*elem* 5 *.X*>). Сегментная форма вычисляется как обычная (*простая*) форма, но затем у ее значения, если это список, отбрасываются внешние скобки, и полученная таким образом последовательность элементов подставляется вместо сегментной формы. Например, если переменная *X* имеет значение (*a*(*b c*)), тогда:

```
(5 .X ())  ⇒ (5(a b c)) ())
(5 !.X ()) ⇒ (5 a (b c) ())
(!.X !.X)  ⇒ (a (b c) a (b c))
        [length e]
```

Значением аргумента этой функции должен быть список. Значение функции — длина этого списка, т.е. число элементов на верхнем уровне списка.

Арифметические функции

```
[+ n1 n2 ... nk]      (k≥2)
```

Значениями аргументов должны быть числа. Значение функции — их сумма.

```
[- n1 n2]
```

Значениями аргументов должны быть числа. Значение функции — их разность.

```
[max n1 n2 ... nk]     (k≥2)
```

Значениями аргументов должны быть числа. Значение функции — их максимум.

```
[min n1 n2 ... nk]      (k≥2)
```

Значениями аргументов должны быть числа. Значение функции — их минимум.

Предикаты

Предикатом называется форма, значение которой рассматривается как логическое значение «истина» или «ложь». При этом в Плэнере «ложью» считается пустой список *()*, а «истиной» — любое другое выражение (чаще всего в роли «истины» выступает атом *T*).

```
[num e]
```

Эта функция-предикат проверяет, является ли числом значение ее аргумента. Если да, то значение функции равно *T* («истина»), иначе – () («ложь»).

```
[empty e]
```

Функция проверяет, является ли значение ее аргумента пустым списком (с любыми скобками). Если да, то значение функции равно *T*, иначе – ().

```
[eq e1 e2]
```

Функция сравнивает значения своих аргументов. Если они равны, то значение функции равно *T*, иначе — ().

```
[neq e1 e2]
```

Аналог, но значения аргументов сравниваются на «не равно».

```
[memb e1 e2]
```

Значением аргумента e_2 должен быть список. Функция проверяет, совпадает ли какой-нибудь элемент (верхнего уровня) этого списка со значением аргумента e_1. Если нет, то значение функции равно () («ложь»), а иначе функция выдает порядковый номер первого из элементов списка, равного значению e_1. (Отметим, что данное число можно трактовать как «истина», поэтому функцию *memb* можно использовать как предикат.)

```
[gt n1 n2], [ge n1 n2], [lt n1 n2], [le n1 n2]
```

Значениями аргументов всех этих функций должны быть числа. Функции сравнивают эти числа на $>$ (*gt*), $\geq$ (*ge*), $<$ (*lt*) и $\leq$ (*le*). При успешном сравнении функции выдают значение *T*, а иначе — значение ().

Логические функции

Так называются функции, реализующие логические операции.

```
[not e]
```

Это «отрицание»: если значение аргумента равно () («ложь»), то функция выдает результат *T* («истина»), а при любом другом значении аргумента — результат ().

```
[and e1 e2 ... ek]     (k≥2)
```

Это «конъюнкция». Функция по очереди вычисляет свои аргументы. Если значение очередного из них равно () («ложь»), то функция, не вычисляя оставшиеся аргументы, заканчивает свою работу со значением (), а иначе переходит к вычислению следующего аргумента. Если функция дошла до вычисления последнего аргумента, то с его значением она и заканчивает свою работу.

```
[or e1 e2 ... ek]     (k≥2)
```

Это «дизъюнкция». Функция по очереди вычисляет свои аргументы. Если значение очередного из них отлично от (), то функция, не вычисляя оставшиеся аргументы, заканчивает свою работу со значением этого аргумента, а иначе переходит к вычислению следующего аргумента. Если функция дошла до вычисления последнего аргумента, то с его значением она и заканчивает свою работу.

```
[cond (p1 e1,1 e1,2 ... e1,k1) ... (pn en,1 en,2 ... en,kn)]
         (n≥1,  ki≥1)
```

Это «условное выражение». Функция последовательно вычисляет первые элементы своих аргументов — предикаты p_i. Если все они имеют значение () («ложь»), тогда функция заканчивает свою работу с этим же значением. Но если нашелся предикат p_i, значение которого отлично от (), т.е. имеет значение «истина», тогда функция уже не будет рассматривать остальные предикаты, а последовательно вычислит формы $e_{i,j}$ из этого i-го аргумента и со значением последней из них закончит свою работу. Отметим, что значения предыдущих форм из этого аргумента нигде не запоминаются, поэтому в качестве этих форм имеет смысл использовать только такие, которые имеют побочный эффект, например, меняют значения переменных.

Блочная и связанные с ней функции

```
[prog (v1 v2 ... vn) e1 e2 ... ek]    (n≥0, k≥1)
```

Эту функцию называют «блочной», поскольку ее вычисление напоминает выполнение блоков в других языках программирования. Вычисление функции начинается с того, что вводятся локальные переменные, перечисленные в ее первом аргументе, и им присваиваются начальные значения: v_i — это либо идентификатор (имя) и тогда вводится локальная переменная с этим именем и без начального значения, либо пара (*id val*) и тогда вводится локальная переменная с именем *id* и начальным значением *val* (выражение *val* при этом не вычисляется). После этого функция последовательно вычисляет остальные свои аргументы — формы e_i, которые принято называть *операторами*. Вычислив последний из них, функция с его значением заканчивает работу, уничтожив при этом свои локальные переменные v_i.

Вычисленные значения всех операторов, кроме последнего, нигде не запоминаются, поэтому имеет смысл в качестве таких операторов использовать только функции с побочным эффектом. Некоторые из этих функций перечислены ниже.

```
[set v e]
```

Это аналог оператора присваивания. Сначала вычисляются оба аргумента, причем значением аргумента *v* должно быть имя переменной, существующей в данный момент. Функция присваивает этой переменной новое значение — значение аргумента *e*. Это же значение является значением функции *set*, но оно, как правило, не используется.

```
[fin v1 v2]
```

Значением обоих аргументов должны быть имена переменных (обозначим их v_1 и v_2), существующих в данный момент, причем переменная v_2 должна иметь значение и им должен быть список. Если этот список пуст, то функция ничего не делает и выдает значение *T*. Иначе функция «расщепляет» список на две части — на его первый элемент, который присваивается переменной 1, и на его «хвост» (т.е. без первого элемента), который становится новым значением переменной v_2 ; в данном случае функция выдает результат ().

```
[return e]
```

Это оператор досрочного выхода из блока. Функция *return* может использоваться только внутри функции *prog*, поскольку она завершает вычисление ближайшей объемлющей блочной функции, объявляя ее значением значение аргумента *e*.

```
[go e]
```

Это оператор перехода. Отметим, что если в качестве оператора функции *prog* указан идентификатор, то он трактуется как метка. Значением аргумента функции *go* должен быть идентификатор — одна из меток ближайшей объемлющей блочной функции. Функция *go* полностью завершает выполнения того оператора этой блочной функции, внутрь которого она входит (на любом уровне), и передает управление на оператор, следующий за этой меткой.

```
[hasval v]
```

Значением аргумента должно быть имя переменной, существующей в данный момент (ею может быть, например, локальная переменная объемлющего блока). Функция проверяет, имеет ли сейчас эта переменная какое-либо значение или нет. Если имеет, то функция возвращает результат *T*, а иначе — результат ().

Сопоставление с образцом

При обработке списков нередко приходится поверять, имеет ли некоторый список определенную структуру. Язык Плэнер учитывает эту потребность и предлагает для такой проверки механизм *сопоставления с образцом*, который реализует встроенная функция *is*:

```
[is pat e]
```

Здесь *pat* — это *образец*, который задает требуемую структуру списка, а значение аргумента *e* — тот список (обозначим его *E*), который проверяется на соответствие данному образцу. Если сопоставление успешно, т.е. *E* соответствует образцу *pat*, то значение функции равно *T* («истина»), а иначе равно () («ложь»).

Как правило, *pat* — это список, причем в простейшем случае его элементами являются *простые образцы*, к которым относятся все простые формы (атомы, обращения к переменным с префиксом «.» и т.д.) и обращения к переменным с префиксом «*» (например, **Y*). В таком случае список *E* соответствует образцу-списку *pat*, если у этих списков равное число элементов и если их элементы попарно соответствуют друг другу (эти соответствия проверяются последовательно слева направо). Образцу, являющемуся формой, соответствует в *E* только элемент, равный значению этой формы. Обращению же к переменной с префиксом «*» соответствует любой элемент в *E*, причем, как побочный эффект сопоставления, этот элемент становится новым значением данной переменной (правда, только при успешном сопоставлении всего образца *pat* со списком *E*).

Например, если переменная *X* имеет значение *b*, то:

```
[is (a .X(*Y d)*Z) (a b (c d) e)] ⇒ T  и Y:= c,  Z:= e,
[is (a *Y d) (a b c d)] ⇒ ()
```

и *Y* не меняет значения.

Переменная с префиксом «*» нужна для того, чтобы указать, что в соответствующем месте анализируемого списка может находиться любой элемент; присваивание же его этой переменной позволяет узнать данный элемент. Отметим также, что значение, присвоенной переменной, может быть использовано в оставшейся части образца-списка:

```
[is (a *X .X a) (a b b a)] ⇒ T
```

и

```
X:= b,
[is (a *X .X a) (a b a a)] ⇒ ()
```

и *X* не меняет значения.

В образце-списке можно также использовать *сегментные образцы*, к которым, в частности, относятся обращения к переменным с префиксом «!*» (например, *!*X*). Переменной с этим префиксом может соответствовать любая последовательность соседних элементов в анализируемом списке, и ей ставится в соответствие та из них, при которой есть соответствие и всем другим элементам образца-списка (если таких последовательностей несколько, то берется самая короткая). Причем, как побочный эффект сопоставления, данная последовательность, заключенная в круглые скобки, становится новым значением переменной (правда, опять же только в случае общего соответствия списка *E* образцу *pat*). Например:

```
[is (a !*Y e) (a b c d e)] ⇒ T
```

и

```
Y:=(b c d) ,
[is (a !*Y d) (a b c b a)] ⇒ T
```

и *Y* не меняет значение,

```
[is (!*Y 1 !*Z) (1 2 3 2 1)] ⇒ T  и Y:=(), Z:=(2 3 2 1) .
```

Режим возвратов

В язык Плэнер встроен т.н. *режим возвратов*, который упрощает реализацию различных поисковых алгоритмов, использующих перебор вариантов. Суть этого режима в следующем. В любом месте программы может быть установлена т.н. *развилка*, от которой возможно несколько вариантов продолжения работы

программы. Выбирается один из вариантов, и программа продолжает свою работу. Если затем окажется, что этот вариант неуспешен, тогда вырабатывается т.н. *неуспех*, по которому программа автоматически «откатывается» к последней (по времени) развилке, отменяя при этом все изменения (в значениях переменных и т.п.), произведенные на неуспешном пути, и в этой развилке выбирается следующий вариант, после чего программа снова «идет вперед». Если в развилке уже не оказалось нерассмотренных альтернатив, то неуспех возвращает программу к предыдущей развилке. (Если развилок в программе уже не осталось, то неуспех «выйдет» на верхний уровень программы — будет завершено вычисление очередного выражения верхнего уровня программы и это вычисление будет считаться неуспешным. Успешно или нет завершилось вычисление очередного выражения программы, далее будет вычисляться следующее выражение.)

В каких местах программы ставить развилки и с какими альтернативами, считать ли выбранный путь вычисления неуспешным и когда вырабатывать неуспех — за все это отвечает автор программы. Встроенный же режим возвратов обеспечивает запоминание мест развилок и то, какие альтернативы в них еще не рассматривались, обеспечивает возврат программы по неуспеху к последней развилке и отмену ранее произведенных изменений в значениях переменных.

Ниже перечислены некоторые из встроенных функций языка Плэнер, позволяющих реализовать режим возвратов.

```
[among e]
```

Значением аргумента должен быть список. Если этот список пуст, функция вырабатывает неуспех, который автоматически возвращает программу к предыдущей ее развилке. Иначе функция запоминает развилку, альтернативами которой является то, что функция в качестве своего значения может выдать любой элемент из этого списка. Вначале функция выдает как свое значение первый элемент списка и завершает на этом работу, после чего программа продолжает свои вычисления. Но если позже в программе возникнет неуспех, который вернет ее вычисление к данной развилке, то функция возобновит свою работу и теперь как свое значение выдаст второй элемент списка, после чего программа снова возобновит свои вычисления от этого места. И так до тех пор, пока в списке остаются нерассмотренные элементы. После выдачи в качестве своего значения последнего элемента списка, функция уничтожает свою развилку.

```
[fail]
```

Эта функция вырабатывает неуспех, по которому программа автоматически возвращается к последней (по времени) развилке.

```
[pset v e]
```

Это аналог функции *set*, т.е. переменной, имя которой является значением аргумента *v*, присваивается новое значение — значение аргумента *e*. Однако, если присваивание, осуществленное функцией *set*, отменяется при неуспехе, то действие функции *pset* при неуспехе не будет отменено. Функция *pset* (и ей подобные) применяется, когда надо сохранить информацию, полученную на неуспешном пути вычисления программы, для последующих путей.

```
[permex  ]
```

Функция осуществляет выход из ближайшей объемлющей функции с именем *f* и объявляет ее результатом значение аргумента *e*. При этом уничтожаются все развилки, появившиеся в программе с начала вычисления этой функции *f*, и «замораживаются» все изменения (в значениях переменных и т.п.), произведенные внутри этой функции. Это значит, что при последующем неуспехе функция *f* уже не будет возобновлять свою работу и ее действия не будут отменены.

ЛИТЕРАТУРА

[**Нильсон73**]
Нильсон Н. Искусственный интеллект. Методы поиска решений: Пер. с англ. — М.: Мир, 1973.

[**Нильсон85**]
Нильсон Н. Принципы искусственного интеллекта: Пер. с англ. — М.: Радио и связь, 1985.

[**Братко**]
Братко И. Программирование на языке Пролог для искусственного интеллекта: Пер. с англ. — М.: Мир, 1990.

[**Лорьер**]
Лорьер Ж.-Л. Системы искусственного интеллекта: Пер. с франц. — М.: Мир, 1991.

[**Слейгл**]
Слейгл Дж. Искусственный интеллект. Подход на основе эвристического программирования: Пер. с англ. — М.: Мир, 1973.

[**Уинстон**]
Уинстон П. Искусственный интеллект: Пер. с англ. — М.: Мир, 1980.

[**Пильщиков**]
Пильщиков В.Н. Язык Плэнер. — М.: Наука, 1983.

[**Семенов**]
Семенов М.Ю. Язык Лисп для персональных ЭВМ. — М.: Издательство МГУ, 1989.