базового множественного предиката *Set of*, а не предиката *set of1* (в Прологе-10 он обозначен как *setof*) наше мнение расходится с мнением Уоррена.

Множественные предикаты являются мощным расширением Пролога. Они могут использоваться (к сожалению, неэффективно) для реализации «отрицания как безуспешного вычисления» и предикатов металогического типа (Kahn, 1984). Если цель *G* не имеет решений, определяемых таким предикатом, как *set_of*, то *notG* истинно. Предикат *var(X)* реализуется проверкой того, имеет ли цель два решения *X = 1; X = 2*. Дальнейшее обсуждение такого поведения множественных предикатов и обзор различных реализаций множественных предикатов можно найти в (Naish, 1985b).

Более подробное описание алгоритма Ли и общей задачи трассировки СБИС имеется в учебниках по СБИС, например в (Breuer и Carter, 1983).

Задача о поиске ключевых слов в контексте была предложена Перлисом как контрольная для сравнения языков программирования высокого уровня. Она была использована для сравнения нескольких языков. Реализацию этой задачи на Прологе мы находим наиболее элегантной из всех известных нам.

Наше описание лямбда-выражений дано под влиянием работы (Warren, 1982a). Такие предикаты, как *apply* и *map_list*, были включены в пакет сервисных программ Пролог-системы Эдинбургского университета. Некоторое время они были в моде, но потом утратили благосклонность пользователей из-за неэффективной компиляции и отсутствия средств преобразования на уровне исходной программы.

Глава 18

Методы поиска

В этой главе рассматриваются программы, охватывающие классические для задач искусственного интеллекта методы поиска. В первом разделе обсуждаются общие понятия и принципы поиска в пространстве состояний для задач, формулируемых в терминах графов пространства состояний. Во втором разделе рассматривается минимаксный алгоритм с альфа-бета отсечением для поиска на дереве игры.

18.1. Поиск на графах пространства состояний

Графы пространства состояний используются для представления задач. Вершины графа являются состояниями задачи. Две вершины графа связаны ребром, если существует некоторое правило перехода (*move*), согласно которому производится преобразование одного состояния в другое. Решение задачи состоит в поиске пути из заданного начального состояния в состояние, соответствующее искомому решению, посредством применения последовательности правил перехода.

Программа 18.1 является базовой для решения задач путем поиска на графах пространства состояний с применением поиска в глубину, описанного в разд. 14.2.

Никаких ограничений для представления состояний вводить не следует. Переходы будем описывать бинарным предикатом *move(State,Move)*, где *Move* - правило перехода, применяемое к состоянию *State*. Предикат *update(State,Move,State1)* используется для поиска состояния *State1*, достижимого с помощью применения правила *Move* к состоянию *State*. В ряде случаев процедуры *move* и *update* целесо-

образно объединять. Здесь они остаются раздельными для простоты изложения и сохранения гибкости и модульности программ, возможно, в ущерб эффективности.

Допустимость возможных переходов оценивается предикатом *legal(State)*, который проверяет, удовлетворяет ли состояние задачи *State* ограничениям задачи. Для того чтобы предупредить зацикливание, программа сохраняет ранее пройденные состояния. Последовательность переходов из начального состояния в конечное строится путем наращивания в третьем аргументе предиката *solve_dfs/3*.

solve_dfs(State,History,Moves) ←
Moves последовательность переходов до достижения требуемого конечного состояния из текущего состояния *State*. *History* содержит ранее пройденные состояния.

```
solve_dfs(State,History,[ ]) ←
    final_state(State).
solve_dfs(State,History,[Move|Moves]) ←
    move(State,Move),
    update(State,Move,State1),
    legal(State1),
    not member(State1,History),
    solve_dfs(State1,[State1|History],Moves).
```

Предложение для тестирования базовой программы

```
test_dfs(Problem,Moves) ←
    initial_state(Problem,State),
    solve_dfs(State,[State],Moves).
```

Программа 18.1. Базовая программа организации поиска в глубину в пространстве состояний.

Чтобы использовать базовую программу организации поиска при решении задачи, программист должен принять решения о представлении состояний и аксиоматизации процедур *move*, *update* и *legal*. Успешность применения базовой программы в значительной степени зависит от выбора подходящего представления состояний.

Допустим, базовая программа применяется для решения известной задачи о волке, козе и капусте. Сформулируем эту задачу неформально. У фермера есть волк, коза и капуста, и все они находятся на левом берегу реки. Фермер должен перевезти это «трио» на правый берег, но в лодку может поместиться что-то одно - волк, коза или капуста. Существенно в этой задаче то, что рискованно оставлять волка вместе с козой (волки неравнодушны к козлятине), равно как и козу с капустой (козы обожают капусту). Фермер всерьез озадачен сложившимся положением и не желает нарушать экологическое равновесие ценой потери пассажира.

Состояния представляются тройкой *wgc(B,L,R)*, где *B* – местонахождение лодки (левый или правый берег), *L* список находящихся на левом берегу, *R* – список находящихся на правом берегу. Начальным и конечным состояниями являются *wgc(left,[wolf,goat,cabbage],[ ])* и *wgc*(right,[ ],[*wolf, goat,cabbage*]) соответственно. На самом деле нет необходимости вести списки «обитателей» правого и левого берегов. Зная, кто в данный момент находится на левом берегу, легко определить обитателей правого берега, и наоборот. Использование двух списков лишь упрощает описание переходов.

Для проверки зацикливания удобно сохранять списки обитателей в отсортированном виде. Таким образом, *волк* всегда будет в списке перед *козой*, и оба они перед капустой, если, конечно, все «пассажиры» находятся на одном берегу.

Переходы из состояния в состояние это перевозка обитателей с одного берега

на другой и поэтому может быть специфицирована конкретным «пассажиром», которого будем называть грузом (*Cargo*). Ситуация, когда фермер переправляется через реку один, специфицируется грузом *alone* (без груза). Недетерминированное поведение предиката *member* позволяет, как показано в программе 18.2, сжато описывать все возможные переходы тремя предложениями: для перевозки с левого берега на правый, для перевозки с правого берега на левый и для одиночного плавания фермера в любом направлении.

Состояния в задаче о волке, козе и капусте представляются структурой вида *wgc (Boat,Left,Right)*, где *Boat* – берег, у которого в данный момент находится лодка, *Left* – список обитателей левого берега, *Right* – список обитателей правого берега.

```
initial state(wgc,wgc(left,[wolf,goat,cabbage], [ ])).

final state(wgc(right,[ ],wolf,goat,cabbage])).

move(wgc(left,L,R),Cargo) ← member(Cargo,L).
move(wgc(right,L,R),Cargo) ← member(Cargo,R).
move(wgc(B,L,R),alone).

update(wgc(B,L,R),Cargo,wgc(B1,L1,R1)) ←
    update_boate(B,B1), update_banks(Cargo,B,L,R,L1,R1).

update boat(left,right).
update boat(right,left).

update_banks(alone,B,L,R,L,R).
update_banks(Cargo,left,L,R,L1,R1) ← select(Cargo,L,L1), insert(Cargo,R,R1).

update_banks(Cargo,right,L,R,L1,R1) ←
    select(Cargo,R,R1), insert(Cargo,L,L1).

insert(X,[Y | Ys],[X,Y | Ys]) ←
    precedes(X,Y).
insert(X,[Y | Ys],[Y | Zs]) ←
    precedes(Y,X), insert(X,Ys,Zs).
insert(X,[ ],[X]).

precedes(wolf,X).
precedes(X,cabbage).

legal(wgc(left,L,R)) ← not illegal(R).
legal(wgc(right,L,R)) ← not illegal(L).

illegal(L) ← member(wolf,L), member(goat,L).
illegal(L) ← member(goat,L), member(cabbage,L).
```

Программа 18.2. Программа для решения задачи о волке, козе и капусте.

Для каждого из указанных переходов должна быть специфицирована процедура, которая изменяла бы местонахождение лодки *update_boat/2* и состав обитателей берегов *update_banks*. Использование предиката *select* позволяет дать компактное описание модифицирующих процедур. Для поддержания списка обитателей берега в упорядоченном состоянии используется процедура *update_banks/3*, облегчающая проверку на зацикливание. В ней учтены все варианты расширения состава обитателей на берегу.

Наконец, должна быть специфицирована проверка допустимости переходов. Существующие здесь ограничения просты. Волк и коза, равно как и коза с капустой, не могут в отсутствие фермера находиться на одном берегу.

Программа 18.2 объединяет в дополнение к базовой программе 18.1 все факты и правила, необходимые для решения задачи о волке, козе и капусте. Ясность программы говорит сама за себя.

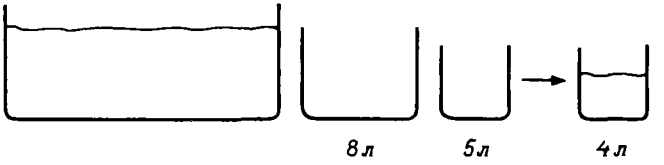

Рис. 18.1. Задача о кувшинах.

Теперь обратимся к применению базовой программы поиска в пространстве состояний для решения другой классической в занимательной математике задачи поиска – задачи о кувшинах с водой. Имеются два кувшина вместимостью 8 и 5 л, и необходимо отмерить 4 л из бочки на 20, а может быть, больше литров. Возможными операциями являются: наполнение кувшина жидкостью из бочки, выливание содержимого кувшина в бочку, переливание из одного кувшина в другой до полного опустошения первого, либо до полного заполнения второго. Рис. 18.1 поясняет суть задачи.

Рассматриваемая задача может быть обобщена на N кувшинов с емкостями $C_1,\ldots,C_N$. Требуется измерить объем V, отличный от любого из C_i, но не превышающий емкости наибольшего кувшина. Решение существует, если величина V кратна наибольшему общему делителю чисел C_i. Для случая двух кувшинов задача имеет решение, поскольку 4 кратно наибольшему общему делителю чисел 8 и 5[1].

Рассмотрим решение этой частной задачи для двух кувшинов произвольной емкости, однако этот подход непосредственно обобщается на любое число кувшинов. Предполагаться, что в базе данных содержатся два факта *capacity(I,CI)* для *I*, равного *1* и *2*. Естественным представлением состояния будет структура *jugs(V1,V2)*, где *V1* и *V2* представляют объемы жидкости, содержащейся в соответствующих кувшинах в текущий момент. Начальное состояние задается структурой *jugs(0, 0)*, а желаемое конечное состояние выражается либо структурой *jugs(0,X)*, либо структурой *jugs(X,0)*, где *X* – искомый объем. Предполагая, что первый кувшин имеет бóльшую емкость, чем второй, достаточно специфицировать только одно конечное состояние *jugs(X,O)*, так как требуемое количество жидкости легко перелить из второго кувшина в первый (сначала освобождается первый кувшин, а затем в него переливается содержимое второго).

Эти данные для решения задачи о кувшинах, объединенные с программой 18.1, дают программу 18.3. В программе определено шесть переходов (предложений *move*): два - для наполнения каждого из кувшинов, два – для освобождения каждого из кувшинов и два – для переливания из одного кувшина в другой. Примером факта, соответствующего заполнению первого кувшина, является *move(jugs(V1,V2),fill(1))*. Явное задание состояний кувшинов позволяет данным «сосуществовать» с данными для решения других задач, например, таких, как задача, представленная программой 18.2. Переходы, связанные с опустошением кувшинов, оптимизированы так, чтобы не опустошать уже пустой кувшин. Модифицирующая процедура, связанная с первыми четырьмя переходами, проста, в то время как операция переливания имеет два варианта. Если общий объем жидкости в кувшинах меньше чем емкость наполняемого кувшина, то опустошаемый кувшин освободится, а наполненный

---

[1] Поскольку 5 и 8 – взаимно простые, то можно отлить не только 4 л, но и 1, 2, ..., 8 л. – *Прим. ред.*

```
initial state(jugs, jugs(0, 0)).
final state(jugs(4, V2)).
final state(jugs(V1, 4)).

move(jugs(V1, V2), fill(1)).
move(jugs(V1, V2), fill(2)).
move(jugs(V1, V2), empty(1)) ← V1 > 0.
move(jugs(V1, V2), empty(2)) ← V2 > 0.
move(jugs(V1, V2), transfer(2, 1)).
move(jugs(V1, V2), transfer(1, 2)).

update(jugs(V1, V2), empty(1), jugs(0, V2)).
update(jugs(V1, V2), empty(2), jugs(V1, 0)).
update(jugs(V1, V2), fill(1), jugs(C1, V2)) ← capacity(1, C1).
update(jugs(V1, V2), fill(2), jugs(V1, C2)) ← capacity(2, C2).
update(jugs(V1, V2), transfer(2, 1), jugs(W1, W2)) ←
    capacity(1, C1),
    Liquid := V1 + V2,
    Excess := Liquid − C1,

    adjust(Liquid, Excess, W2, W1).

adjust(Liquid, Excess, Liquid, 0) ← Excess ≤ 0.
adjust(Liquid, Excess, V, Excess) ← Excess > 0, V := Liquid − Excess.

legal(jugs(V1, V2)).

capacity(1, 8).
capacity(2, 5).
```

Программа 18.3. Программа для решения задачи о кувшинах.

будет содержать весь объем жидкости. В противном случае наполняемый кувшин будет залит полностью, а в опустошаемом кувшине сохранится остаток, равный разности между общим объемом жидкости и емкостью наполненного кувшина. Рассмотренные действия реализуются предикатом *adjust/4*. Заметим, что проверка допустимости переходов тривиальна, так как все достижимые состояния допустимы.

Наиболее интересные задачи имеют слишком большое пространство поиска, которое не под силу такой программе, как программа 18.1. Одна из возможностей усовершенствования процедуры поиска состоит в привлечении бóльших знаний для выполнения переходов. Решения задачи о кувшинах могут быть найдены посредством наполнения одного из кувшинов, когда это возможно, освобождения другого кувшина, когда это возможно, и в противном случае переливания содержимого наполненного кувшина в опустошенный кувшин. Тем самым вместо шести переходов потребуется специфицировать только три, а поиск будет более направленным, поскольку только один переход будет применим в любом данном состоянии. Правда, этот путь может не привести к оптимальному решению, если ошибочно выбирается постоянно наполненный кувшин.

Развивая это соображение, можно эти три перехода объединить в один переход более высокого уровня *fill and transfer*. В обсуждаемой тактике предполагается наполнение одного кувшина и переливание всего его содержимого в другой кувшин с освобождением последнего по мере необходимости. Следующий фрагмент программы соответствует переливанию жидкости из большего по емкости кувшина в кувшин меньшей емкости.

```
move(jugs(V1, V2), fill and transfer(1)).

update(jugs(V1, V2), fill and transfer(1), jugs(0, V)) ←
```

```
capacity(1,C1),
capacity(2,C2),
C1 > C2,
V := (C1 + V2) mod C2.
```

Для решения задачи, представленной на рис. 18.1, в этом случае необходимо выполнить только три операции наполнения и переливания из одного кувшина в другой.

Привлечение знаний проблемной области приводит к полному изменению описания задачи и программированию, возможно, на другом уровне.

Другая возможность улучшения эффективности процедуры поиска, исследованная в ранних работах по искусственному интеллекту, связана с эвристическими методами. Общая схема основана на явном выборе следующего состояния при поиске на графе пространства состояний. Этот выбор зависит от численных меток, присвоенных позициям. Метка, вычисляемая с помощью оценочной функции, является мерой качества позиции. Поиск в глубину можно рассматривать как частный случай процедуры поиска, в которой значение оценочной функции равно расстоянию от текущего состояния до начального состояния, в то время как в процедуре поиска в ширину оценочная функция имеет значение, обратное указанному расстоянию.

Рассмотрим два метода поиска, в которых явно используется оценочная функция: «подъем на холм», и «сначала лучший». Для оценочной функции введем предикат *value(State,Value)*. Методы будут описаны абстрактно.

Метод «подъем на холм» является обобщением поиска в глубину, в котором следующей выбирается позиция с наивысшей оценкой, а не самая левая позиция, как предопределено Прологом. Верхний уровень процедуры поиска, реализованной программой 18.1, сохраняется. В методе «подъем на холм» предикат *move* генерирует все состояния, которые могут быть достигнуты из текущего состояния за один переход с помощью предиката *set of*. Затем эти состояния выстраиваются в порядке убывания вычисленных значений оценочной функции. Предикат *evaluate and order(Move,State,MVs)* устанавливает связь между упорядоченным списком *MVs* пар «переход значение оценочной функции» и списком переходов *Moves* из состояния *State*. Описанный метод поиска реализуется программой 18.4.

```
solve hill climb(State,History,Moves) ←
    Moves  последовательность переходов для достижения искомого конечного состояния из текущего состояния State.
    History  ранее пройденные состояния.

solve hill climb(State,History,[ ]) ←
    final state(State).
solve hill climb(State,History,Move|Moves]) ←
    hill climb(State,Move),
    update(State,Move,State1),
    legal(State1),
    not member(State1,History),
    solve hill climb(State1,[State1|History],Moves).

hill climb(State,Move) ←
    set of(M, move(State,M),Moves),
    evaluate and order(Moves,State,[ ],MVs),
    member((Move,value),MVs).
evaluate and order(Moves,State,SoFar,OrderedMVs) ←
    Все переходы Moves из текущего состояния State оцениваются и упорядочиваются, результат в OrderedMoves.
    SoFar  накопитель для частичных вычислений.
```

```
evaluate and order([Move|Moves],State,MVs,OrderedMVs) ←
    update(State,Move,State1),
    value(State1,Value),
    insert((Move,Value),MVs,MVs1),
    evaluate_and_order(Moves,State,MVs1,OrderedMVs).
evaluate_and_order([ ],State,MVs,MVs).
insert(Mv,[ ],[MV]).
insert((MV),[(M1,V1)|MVs],[(M,V),(M1,V1)|MVs]) ←
    V ≥ V1.
insert((M,V),[(M1,V1)|MVs],[(M1,V1)|MVs1] ←
    V < V1,insert((M,V),MVs,MVs1).
```

Предложение для тестирования базовой программы

```
test_hill climb(Problem,Moves) ←
    initial state(Problem,State),Solve(State,[State],Moves).
```

Программа 18.4. Базовая программа для решения задачи поиска методом «подъем на холм».

Чтобы познакомиться с работой программы, воспользуемся примером дерева из программы 14.8, добавив к нему факты, определяющие значения оценочной функции для каждого перехода. Необходимые для тестирования данные собраны в программе 18.5. Объединение программ 18.4 и 18.5 вместе с необходимыми определениями предикатов *update* и *legal* обеспечивает поиск по дереву, в процессе которого были выбраны вершины *d* и *j*. Эту программу легко протестировать на задаче о волке, козе и капусте, используя в качестве оценочной функции число «обитателей» на правом берегу реки.

Программе 18.4 присущ недостаток, состоящий в повторном вычислении оценочной функции: при достижении после перехода *Move* некоторого состояния она вычисляется для выбора нового перехода, а затем повторно – при выполнении предиката *update*. Повторного вычисления можно избежать введением дополнительного аргумента в отношение *move* и сохранением состояния и перехода вместе с вычисленным значением до упорядочения переходов. Другая возможность оптимизации вычислений для одного и того же перехода состоит в использовании функции запоминания. Выбор эффективного метода зависит от конкретной задачи. Для задач с простой процедурой *update* представленная здесь программа будет наилучшей.

«Подъем на холм» полезен, когда в пространстве состояний есть только один «холм» и оценочная функция является действительным указателем направления поиска. Существенно, что метод осуществляет локальный просмотр графа пространства состояний, принимая решение о направлении перехода лишь на основе текущего состояния.

Альтернативный метод поиска «сначала – лучший» основан на глобальном рассмотрении полного пространства состояний. Лучшее состояние выбирается из всех не пройденных до сих пор состояний.

Программа 18.6 для поиска «сначала – лучший» является обобщением алгоритма поиска в ширину, рассмотренного в разд. 17.2. На каждом этапе поиска выполняется очередной наилучший из доступных переходов. Для удобства сравнения программа 18.6 написана, насколько это оказалось возможным, в стиле программы 18.4 для поиска методом «подъем на холм».

На каждом этапе поиска рассматривается не один, а множество возможных переходов. Об этом свидетельствует множественное число имен предикатов *updates* и *legals*. Так, с помощью предиката *legals(States,States1)* производится фильтрация

```
initial_state(tree,a). value(a,0). final_state(j).

move(a,b).    value(b,1).    move(c,g).    value(g,6).
move(a,c).    value(c,5).    move(d,j).    value(j,9).
move(a,d).    value(d.7).    move(e,k).    value(k,1).
move(a,e).    value(e,2).    move(f,h).    value(h,3).
move(c,f).    value(f,4).    move(f,i).    value(i,2).
```

Программа 18.5. Тестовые данные.

solve_best(Frontier,History,Moves) ←
Moves последовательность переходов для достижения искомого конечного состояния из начального состояния. *Frontier* содержит подлежащие рассмотрению текущие состояния. *History* содержит ранее пройденные состояния.

```
solve_best([state(State,Path,Value)|Frontier],History,Moves) ←
    final state(State),reverse(Path,Moves).
solve_best([state(State,Path,Value)|Frontier],History,FinalPath) ←
    set of(M,move(State,M),Moves),
    updates(Moves,Path,State,States),
    legals(States,States1),
    news(States1,History,States2),
    evaluates(States2,Values),
    inserts(Values,Frontier,Frontier1),
    solve_best(Frontier1,)[State|History],FinalPath).
```

```
updates(Moves,Path,State,States) ←
    States список возможных состояний, достижимых из текущего состояния State,
    согласованный со списком возможных переходов Moves. Path – путь из начального
    состояния к состоянию State.
updates([M|Ms],Path,S,[(S1,[M|Path])|Ss]) ←
    update(S,M,S1), updates(Ms,Path,S,Ss).
updates([ ],Path,State,[ ]).
```

legals(States,States1) ←
States1 – подмножество списка допустимых состояний *States*.

```
legals([(S,P)|States],[(S,P)|States1]) ←
    legal(S), legals(States,States1).
legals([(S,P)|States],States1) ←
    not legal(S), legals(States,States1).
legals([ ],[ ]).
```

news(States,History,States1) ←
States список состояний из *States*, не входящих в список *History*.

```
news([(S,P)|States],History,States1) ←
    member(S,History),news(States,History,States1).

news([(S,P)|States],History,[(S,P)|States1]) ←
    not member(S,History), news(States,History,States1).
news([ ],History,[ ]).
```

evaluates(States,Value) ←
Values – список наборов из *States*, расширенных оценками состояний.

```
evaluates([(S,P)|States],[state(S,P,V)|Values]) ←
    value(S,V), evaluates(States,Values).
evaluates([ ],[ ]).
```

Программа 18.6. Базовая программа для решения задачи поиска методом «сначала – лучший».

```
inserts(States,Frontier,Frontier1) ←
    Frontier1  результат включения состояний States в текущую границу Frontier.

    inserts([Value|Values],Frontier,Frontier1) ←
        insert(Value,Frontier,Frontier0),
        inserts(Values,Frontier0,Frontier1).
    inserts([ ],Frontier,Frontier).

    insert(State,[ ],State]).
    insert(State,[State1|States],State,[State1|States]) ←
        less_than(State1,State).
    insert(State,[State1|States],[State|States]) ←
    equals(State,State1).
insert(State,[State1|States],[State1|States1]) ←
    less_than(State,State1),insert(State,States,States1).

equals(state(S,P,V),state(S,P1,V)).

less_than(state(S1,P1,V1),state(S2,P2,V2)) ← S1 ≠ S2, V1 < V2.
```

Программа 18.6. (Продолжение).

```
solve_best(Frontier,History,Moves) ←
    Moves  последовательность переходов для достижения искомого конечного состояния из начального состояния. Frontier содержит текущие состояния, подлежащие рассмотрению. History содержит ранее пройденные состояния.

solve_best([state(State,Path,Value)|Frontier],History,Moves) ←
    final_state(State),reverse(Path,[ ],Moves).
solve_best([state(State,Path,Value)|Frontier],History,FinalPath) ←
    set_of(M,move(State,M),Moves),
    update_frontier(Moves,State,Path,History,Frontier,Frontier1),
    solve_best(Frontier1,[State|History],FinalPath).

update_frontier([M|Ms],State,Path,History,F,F1) ←
    update(State,M,State1),
    legal(State1),
    value(State1,Value),
    not_member(State1,History),
    insert((State1,[M|Path],Value),F,F0),
    update_frontier(Ms,State,Path,History,F0,F1).
update_frontier([ ],S,P,H,F,F).

insert(State,Frontier,Frontier1) ← См. программу 18.6.
```

Программа 18.7. Более эффективная базовая программа для решения задачи поиска методом «сначала лучший».

множества последующих состояний путем их проверки на соответствие ограничениям задачи. Один из недостатков алгоритма поиска в ширину (а следовательно, и поиска «сначала лучший») заключается в неудобстве вычисления прилагаемого пути. Каждое состояние должно быть явно запомнено вместе с пройденным путем. Эта особенность отражена в программе.

Программа 18.6 оттестирована на данных, содержащихся в программе 18.5; порядок прохождения вершин здесь тот же, что и в случае применения метода «подъем на холм».

В программе 18.6 каждый шаг процесса выполняется явно. На практике программу можно сделать более эффективной, объединяя некоторые шаги. Например, при фильтрации сгенерированных состояний можно производить одновременную проверку новизны и допустимости состояния, что позволяет обходиться без обра-

зования промежуточных структур данных. Программа 18.7 иллюстрирует идею объединения всех проверок в одной процедуре *update frontier*.

Упражнения к разд. 18.1

1. Переделайте программу для задачи о кувшинах, основываясь на двух операциях «наполнить и перелить».

2. Напишите программу для решения задачи о миссионерах и каннибалах, которая формулируется следующим образом:

Три миссионера и три каннибала находятся на левом берегу реки. Здесь же небольшая лодка, вмещающая не более двух человек. Все хотят перебраться на другой берег. Если на каком-либо берегу миссионеров окажется больше, чем каннибалов, то миссионеры обратят каннибалов в свою веру. Найти последовательность ездок, гарантирующую безопасность миссионерам и свободу вероисповедания каннибалам.

3. Пять ревнивых мужей (Э. Дьюдени, 1917).

Во время наводнения пять супружеских пар оказались отрезанными от суши водой. В их распоряжении была одна лодка, которая могла одновременно вместить только трех человек. Каждый супруг был настолько ревнив, что не мог позволить своей супруге находиться в лодке или на любом берегу с другим мужчиной (или мужчинами) в его отсутствие. Найти способ переправить на сушу этих мужчин и их жен в целости и сохранности.

4. По аналогии с программой 18.1 разработайте базовую программу для решения задач методом поиска в ширину, основываясь на программах разд. 17.2.

5. Используйте разработанную вами базовую программу для решения головоломки о восьми ферзях. Найдите подходящую оценочную функцию.

18.2. Игровые деревья поиска

Что происходит, когда мы играем в какую-нибудь игру? Игра начинается, например, с расстановки шахматных фигур, сдачи карт, распределения спичек и т. п. Решив, кто начнет игру, игроки делают ходы по очереди. После каждого хода позиция игры соответственно обновляется.

Превратим это туманное описание в простую базовую программу игры. Предложением верхнего уровня будет:

```
играть(Игра,Результат) ←
    инициализировать(Игра,Позиция,Игрок),
    отобразить(Позиция,Игрок),
    играть(Позиция,Игрок,Результат).
```

Предикат *инициализировать(Игра,Позиция,Игрок)* определяет начальную позицию *Позиция* игры *Игра* и игрока *Игрок*, начинающего игру.

Игра представляется последовательностью шагов, каждый из которых состоит в выборе игроком хода, выполнении хода и определении игрока, который должен выполнять следующий ход. Поэтому более точное описание дает процедура *играть*, имеющая три аргумента: позиция игры, игрок, выполняющий ход, и конечный результат. При этом удобно отделить выбор хода *выбрать_ход/3* от его выполнения *ходить/3*. Остальные предикаты в представленном ниже предложении служат для отображения состояния игры и определения игрока, выполняющего следующий ход:

```
играть(Позиция,Игрок,Результат) ←
    выбрать_ход(Позиция,Игрок,Ход),
    ходить(Ход,Позиция,Позиция1),
    отобразить_игру(Позиция1,Игрок),
    следующий_игрок(Игрок,Игрок1),
    !, играть(Позиция1,Игрок1,Результат).
```

Программа 18.8 определяет логическую основу игровых программ. Используя ее для написания программ конкретных игр, следует сосредоточить внимание на