# ЭЛЕМЕНТЫ КОМБИНАТОРИКИ

***ЦЕЛЬ***: Познакомиться с реализацией основных комбинаторных схем на языке пролог и научиться писать программы по их генерации.

## 1.1. ПРОГРАММА ВЫЧИСЛЕНИЯ ПЕРЕСТАНОВОК БЕЗ ПОВТОРЕНИЙ

Перестановка из n-элементного множества М есть упорядоченный набор длины n, составленный из попарно различных элементов множества М. Обозначим через PM множество всех перестановок из n элементов и через Pn число всех перестановок из n элементов. Например, если M={a,b,c}, то PM={(a,b,c), (a,c,b), (b,a,c), (b,c,a), (c,a,b), (c,b,a)}; Pn=6.

Процедура **permulation(A,L)** строит список L всех перестановок алфавита А. Например, если А=[a,b,c], то

L=[[a,b,c],[b,a,c],[b,c,a],[a,c,b],[c,a,b],[c,b,a]].

Процедура **permut(A,P)** выдает перестановку P алфавита А. Например, если А=[a,b,c], то программа выдаст:

P=[a,b,c] →;

P=[b,a,c] →;

P=[b,c,a] →;

P=[a,c,b] →;

P=[c,a,b] →;

P=[c,b,a] →;

no

Процедура **insert(X,P,L)** символ X “пропускает” через список L и выдает результат P. Например, если X=a, L=[1,2,3], то результат будет следующий:

P=[a,1,2,3] →;

P=[1,a,2,3] →;

P=[1,2,a,3] →;

P=[1,2,3,a] →;

no

Текст программы:

**d(A,L):-permulation(A,L).**

**permulation(A,L):-findall(P,permut(A,P),L).**

**permut([],[]).**

**permut([X|L],P):-permut(L,L1),insert(X,P,L1).**

**insert(X,[X|T],T).**

**insert(X,[Y|T],[Y|T1]):-insert(X,T,T1).**

**insert(X,[],[X]).**

Встроенный предикат **findall(****Template, Goal, Bag)** принимает значение успешно, если Bag – есть список всех тех переменных из спискаTemplate, для которых цель Goal успешна. Список Bag не упорядочивается, повторы элементов не устраняются. Если объектов удовлетворяющих цели нет, то запрос истинен, если список Bag пуст.

**1.2. ПРОГРАММА ВЫЧИСЛЕНИЯ СОЧЕТАНИЙ БЕЗ ПОВТОРЕНИЙ**

Сочетание из n элементов по r есть r – элементное подмножество n – элементного множества M.

Обозначим через CrM множество всех сочетаний из n элементов по r и через Crn число всех сочетаний из n элементов по r. Например, если M={a,b,c}, то C1M={(a),(b),(c)},C2M={(a,b),(a,c),(b,c)};C13=|C1M|=3;C23=|C2M|=3.

Процедура **combination(A,R,C)** строит множество С всех сочетаний элементов алфавита A по R элементов в каждом сочетании.

Процедура **combin2(A,C1)** строит множество С всех сочетаний элементов алфавита А по 2 элемента в каждом сочетании.

Процедура **combinK(C1,A,R,K,C)**, исходя из множества С1 всех сочетаний элементов алфавита А по 2, последовательно строит множества сочетаний по K=3,4,…,R элементов.

Процедура **combinK\_K1(C1,A,C),** исходя из множества C1 всех сочетаний элементов алфавита А по K, строит множество всех сочетаний элементов алфавита A по K+1. Вычисление осуществляются с помощью процедуры **comb(C1,A,Ac,C),** содержащую переменную-накопитель Ac. Пусть, например, C1=[[a,b,c],[a,b,d],[a,d,e],[a,c,d],[a,c,e],[a,d,e],[b,c,d],[b,c,e],[b,d,e],[c,d,e]], A=[a,b,c,d,e].

Берем элемент а из А. собираем в С1 все сочетания, содержащие элемент а. выбрасываем их из С1 и к каждому сочетанию результата С2=[[b,c,d],[b,c,e],[b,d,e],[c,d,e]] приписываем спереди элемент а. Результата помещаем в накопитель Ac. Берем элемент b из А и проделываем аналогичную операцию для С2 и b. Результат приписываем в накопителе Ac. И так далее до исчерпания множества C1. В накопителе Ас будут собраны все сочетания из А по 4.

**d(A,R,C):-A=[a,b,c,d,e],combination(A,R,C).**

**combination(A,R,C):-length(A,L),(R=0;R>L),C=[].**

**combination(A,R,C):-length(A,L),R=1,C=A.**

**combination(A,R,C):-length(A,L),R\=0,R\=1,R<L+1,combin2(A,C1),combinK(C1,A,R,2,C).**

**combin2(A,C1):-combin2\_rep(A,[],C1).**

**combin2\_rep([H|T],Ac,C):-findall([H,X],member(X,T),L),**

**append(Ac,L,L1),combin2\_rep(T,L1,C).**

**combin2\_rep([],C,C).**

**combinK(C1,A1,R,K,C):-K1 is K+1,K1=<R,**

**combinK\_K1(C1,A1,L),combinK(L,A1,R,K1,C).**

**combinK(C,A1,R,R,C).**

**combinK\_K1(C1,A,C):-comb(C1,A,[],C).**

**comb(C1,[H|A1],Ac,C):-**

**findall(X,(member(X,C1),member(H,X)),L),difset(C1,L,C2),**

**findall(X,(member(Y,C2),add(H,Y,X)),C1H),**

**append(Ac,C1H,Ac1),comb(C2,A1,Ac1,C).**

**comb([],A1,C,C).**

**add(X,L,[X|L]).**

**difset(X,Y,T):-dif1(X,Y,X,T).**

**dif1([R|X],Y,[R|Z],T):-not(member(R,Y)),**

**append(Z,[R],Z1),dif1(X,Y,Z1,T).**

**dif1([R|X],Y,[R|Z],T):-member(R,Y),dif1(X,Y,Z,T).**

**dif1([],Y,Z,Z).**

**append([],L,L).**

**append([X|L1],L2,[X|L3]):-append(L1,L2,L3).**

**member(X,[X|Y]).**

**member(X,[Y|Z]):-member(X,Z).**

Разберитесь в программе и проверьте ее правильность задав вопросы. Что бы узнать все сочетания множества [a,b,c,d,e] по 4 необходимо задать вопрос:

**?- d(A,4,R).**

**1.3. ПРОГРАММЫ ВЫЧИСЛЕНИЯ РАЗМЕЩЕНИЙ БЕЗ ПОВТОРЕНИЙ**

Размещение из n элементов по r есть упорядоченный набор, состоящий из r Попарно различных элементов, взятых из n-элементного множества M.

Обозначим через ArM множество всех размещений из n элементов по r и через Arn число всех размещений из n элементов по r. Заметим, что для n-элементного множества M множество перестановок PM=AnM, а их число Pn=Ann.

Пример. M={a,b,c}; A1M={(a),(b),(c)}; A2M={(a,b),(a,c),(b,c),(b,a),(c,a),(c,b)}; A31=|A1M|=3; A23=|A2M|=6.

Процедура **allocation(A,R,Res)** вычисляет множество Res размещений элементов алфавита A по R элементов в каждом размещении.

Процедуры combination и permutation вычисляют множества сочетаний и перестановок; подробнее о них сказано в соответствующих программах.

Процедура **comb\_permut(C,R)** по сочетанию C строит множество R всех его перестановок.

**d(A,R,Res):-A=[a,b,c,d,e],allocation(A,R,Res).**

**allocation(A,R,Res):-length(A,L),(R=0;R>L), Res=[].**

**allocation(A,R,Res):-length(A,L),R=1,Res=A.**

**allocation(A,R,Res):-length(A,L),R\=0,R<L+1,R\=1,combination(A,R,C), comb\_permut(C,Res).**

**comb\_permut(C,R):-comb\_permut1(C,[],R).**

**comb\_permut1([H|T],Ac,R):-permulation(H,PH),**

**append(Ac,PH,Ac1),comb\_permut1(T,Ac1,R).**

**comb\_permut1([],R,R).**

**permulation(A,L):-findall(P,permut(A,P),L).**

**permut([],[]).**

**permut([X|L],P):-permut(L,L1),insert(X,P,L1).**

**insert(X,[X|T],T).**

**insert(X,[Y|T],[Y|T1]):-insert(X,T,T1).**

**insert(X,[],[X]).**

**combination(A,R,C):-combin2(A,C1),combinK(C1,A,R,2,C).**

**combin2(A,C1):-combin2\_rep(A,[],C1).**

**combin2\_rep([H|T],Ac,C):-findall([H,X],member(X,T),L),**

**append(Ac,L,L1),combin2\_rep(T,L1,C).**

**combin2\_rep([],C,C).**

**combinK(C1,A1,R,K,C):-K1 is K+1,K1=<R,**

**combinK\_K1(C1,A1,L),combinK(L,A1,R,K1,C).**

**combinK(C,A1,R,R,C).**

**combinK\_K1(C1,A,C):-comb(C1,A,[],C).**

**comb(C1,[H|A1],Ac,C):-**

**findall(X,(member(X,C1),member(H,X)),L),difset(C1,L,C2),**

**findall(X,(member(Y,C2),add(H,Y,X)),C1H),**

**append(Ac,C1H,Ac1),comb(C2,A1,Ac1,C).**

**comb([],A1,C,C).**

**add(X,L,[X|L]).**

**difset(X,Y,T):-dif1(X,Y,X,T).**

**dif1([R|X],Y,[R|Z],T):-not(member(R,Y)),**

**append(Z,[R],Z1),dif1(X,Y,Z1,T).**

**dif1([R|X],Y,[R|Z],T):-member(R,Y),dif1(X,Y,Z,T).**

**dif1([],Y,Z,Z).**

**append([],L,L).**

**append([X|L1],L2,[X|L3]):-append(L1,L2,L3).**

**member(X,[X|Y]).**

**member(X,[Y|Z]):-member(X,Z).**

***ЗАДАНИЕ 1.1***

Напишите программу поиска всех перестановок с повторениями

***ЗАДАНИЕ 1.2***

Напишите программу поиска всех сочетаний с повторениями

***ЗАДАНИЕ 1.3***

напишите программу поиска всех размещений с повторениями.

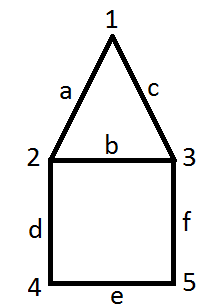
# АЛГОРИТМЫ НА ГРАФАХ И ИХ ПРОЛОГ ПРОГРАММЫ

***ЦЕЛЬ:*** Дать представление об основных задачах на графах.

В настоящее время возросла роль графовых моделей в различных практических приложениях таких, как автоматизированные системы управления, экспертные системы, системы искусственного интеллекта и т.п.

Графом называют систему объектов G = (V,E), где V={V1,...,Vm} - множество вершин, а E = {E1 = (Vi1,Vj1),...,En = (Vin,Vjn)} - множество ребер.

Если пара (Vik,Vjk) упорядочена, то граф называют ориентированным, в противном случае граф - неориентированный или просто граф.

Например,

V={1,2,3,4,5}

E={a=(1,2),

b=(2,3),

c=(1,3),

d=(2,4),

e=(4,5),

f=(3,5)}

Граф в прологе удобно задавать с помощью предиката 'ребро'. В общем случае этот предикат выглядит следующим образом

ребро(Имя\_ребра, Вершина\_начало\_ребра, Вершина\_конец\_ребра, Вес\_ребра).

В частных случаях некоторые аргументы предиката 'ребро' могут отсутствовать. Так для графа, нарисованного выше, имеем

**ребро(a,1,2).**

**ребро(b,2,З).**

**ребро(c,1,З).**

**ребро(d,2,4).**

**ребро(e,4,5).**

**ребро(f,З,5).**

Маршрут из вершины Vi в Vj - чередующаяся последовательность вершин и ребер, начинающаяся в вершине Vi и заканчивающаяся

в вершине Vj и в которой любая пара соседних элементов инцидентна

( вершина V и ребро E инцидентны, если E = (V,W) или E = (W,V)).

Рассмотрим задачу поиска эйлеровой цепи и эйлерового цикла.

Эйлерова цепь - маршрут, содержащий все ребра графа в точности

один раз. Эйлеров цикл - эйлерова цепь, у которой начальная и ко-

нечная вершины совпадают.

Напишем правила для поиска эйлеровой цепи в графе, описав предикат

найти\_путь(Текущая,Пройденные),

где первый аргумент - Текущая - вершина, а второй - список Пройденные, хранящий номера пройденных в процессе поиска ребер.

Правило 1: Если найден путь, длина которого равна общему количеству

ребер, то искомый путь найден и его надо напечатать.

**найти\_путь(Текущая,Пройденные):- длина(Пройденные,6),**

**write(Пройденные).**

Правило 2: Если остались еще не пройденные ребра, то надо взять

ребро, исходящее из текущей вершины и не принадлежащее списку

пройденных.

**найти\_путь(Текущая,Пройденные):-**

**длина(Пройденные,N),N<6, ребро(Ребро,Текущая,Новая),**

**не\_принадлежит(Ребро,Пройденные),**

**найти\_путь(Новая,[Ребро|Пройденные]).**

(предикат 'не\_принадлежит' написать самостоятельно).

***ЗАДАНИЕ 2.1***

Напишите самостоятельно предикаты ‘длина’ и ‘не\_принадлежит’. Задайте запрос на поиск эйлеровой цепи. Запрос на поиск эйлеровых цепей, начинающихся из вершины \_Начальная будет иметь вид:

**?-найти\_путь(\_Начальная,[]).**

***ЗАДАНИЕ 2.2***

Написать программу поиска эйлеровых циклов.

***ЗАДАНИЕ 2.3***

Гамильтонава цепь - маршрут, в котором содержатся все вершины графа ровно один раз. Написать программу поиска гамильтоновой цепи в графе.

***ЗАДАНИЕ 2.4***

Гамильтонов цикл - гамильтонова цепь, у которой начальная и конечная вершины совпадают. Написать программу поиска гамильтонова цикла в графе.

***ЗАДАНИЕ 2.5***

Написать программу поиска всех вершин, достижимых из данной вершины. Вершина V достижима из вершины W, если существует путь из вершины W в вершину V.

***ЗАДАНИЕ 2.6***

Граф называют связанным, если после снятия ориентации в нем для любой пары вершин существует путь, их соединяющий. Написать программу, определяющую связан ли граф.

***ЗАДАНИЕ 2.7***

Решите одну из следующих задач, согласно порядковому номеру в группе:

1. Компонентой связности графа называют связанный подграф графа, не являющийся часть другого связанного подграфа. Написать программу печати компонент связанности.

2. Множество W вершин неориентированного графа G называется внешне устойчивым, если для любой вершины V1 из множества V(G)\W найдется вершина V2 из W такая, что (V1,V2) - ребро графа G. Написать программу поиска наименьшего внешне устойчивого множества графа.

3. Множество W вершин неориентированного графа G называется внутренне устойчивым, если для любой пары вершины V1 и V2 из W не существует ребра (V1,V2) в графе G. Написать программу поиска наибольшего внутренне устойчивого множества графа.

4. Построить остов в неориетированном связанном графе. Остовом называют дерево, построенное из ребер графа и содержащее все его вершины.

5. В графе, ребрам которого приписаны веса, написать программу поиска пути максимального веса, соединяющий заданную пару вершин. Вес пути равен сумме весов ребер, входящих в этот путь.

**3. СТРАТЕГИИ ПОИСКА ПУТИ НА ГРАФЕ**

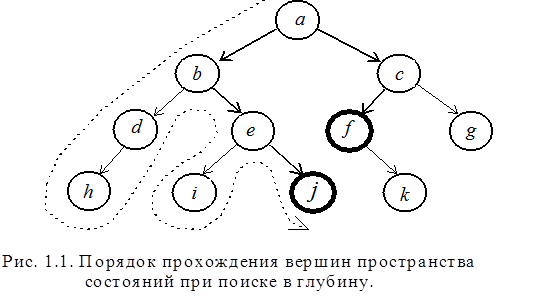
***ЦЕЛЬ:*** Познакомится с классическим алгоритмом обхода деревьев широко используемых в задачах искусственного интеллекта и научиться применять их на практике.

**3.1. СТРАТЕГИЯ ПОИСКА В ГЛУБИНУ**

Логический вывод по существу сводится к достижению цели G на основе последовательного процесса доказательства истинности (или ложности) частичных целей через механизм унификации и возвратов.

Существуют различные подходы к проблеме поиска решающего пути для задач, сформулированных в терминах пространства состояний.

Пространство состояний – это граф, вершины которого соответствуют ситуациям, встречающимся в задаче, а решение задачи сводится к поиску пути на графе. Вершины графа соответствуют проблемным ситуациям, дуги – разрешенным переходам из одних состояний в другие. Задача отыскания плана решения задачи эквивалентна задаче построения пути между заданной начальной ситуацией и некоторой указанной заранее конечной ситуацией, называемой целевой вершиной. В терминах пространства ситуаций основными стратегиями поиска являются стратегии поиска в глубину и в ширину. Рассмотрим сначала первую из них (рис. 1.1).



В терминах логического программирования вычисление цели j приводит к полному обходу в глубину конкретного дерева поиска цели j, в котором выбирается самая левая цель. По существу здесь декларирован некоторый порядок извлечения целей. Действительно, говоря о поиске в глубину, мы имеем в виду тот порядок, в котором рассматриваются альтернативы в пространстве состояний. В соответствии с этим порядком запоминается одна альтернативная вершина и связанные с ней потомки. Сама система Prolog при выполнении цели проверяет цели по принципа поиска в глубину.

Весь проход в глубину образует путь, который, возможно, не достигнет цели, поэтому сработает механизм возврата для поиска альтернативы, вновь ведущей вглубь.

Рассмотрим программу поиска на графе в глубину на языке Prolog:

after – состояние дуг исходного графа

but – целевые вершины.

**after(a,b).**

**after(a,c).**

**after(b,d).**

**after(b,e).**

**after(c,f).**

**after(c,g).**

**after(d,h).**

**after(e,i).**

**after(e,j).**

**after(f,k).**

**but(f).**

**but(j).**

**solve(N,S):-**

**long([],N,S).**

**long(P,N,[N|P]):-**

**but(N).**

**long(P,N,S):-**

**after(N,N1),**

**not(member(N1,P)),**

**long([N|P],N1,S).**

**member(X,[X|Q]).**

**member(X,[H|Q]):-member(X,Q).**

Для поиска пути из вершины а в целевые вершины необходимо ввести вопрос **?-solve(a,X).**

В ответ будет выдано 2 ответа:

**X=[j,e,b,a]->;**

**X=[f,c,a]->;**

**false.**

**3.2. СТРАТЕГИЯ ПОИСКА В ШИРИНУ**

В противоположность поиску в глубину стратегия поиска в ширину предусматривает переход в первую очередь к вершинам, ближайшим к стартовой вершине. В результате процесс поиска имеет тенденцию развиваться более в ширину, чем в глубину, что иллюстрирует рис. 1.2.



Поиск в ширину алгоритмизируется не так легко, как поиск в глубину. Причина состоит в том, что приходится сохранять все множество альтернативных вершин-кандидатов, а не только одну вершину, как при поиске в глубину. Кроме того, если необходимо в процессе поиска получить решающий путь, то одного множества вершин не достаточно. Поэтому хранят множество путей-кандидатов. Для того, чтобы выполнить поиск в ширину при заданном множестве путей-кандидатов, нужно:

если голова первого пути – это целевая вершина, то взять путь в качестве решения, иначе

удалить первый путь из множества кандидатов и породить множество всех возможных продолжений этого пути на один шаг; множество продолжений добавить в конец множества кандидатов, а затем выполнить поиск в ширину с полученным новым множеством.

В случае примера рис. 1.2 этот процесс будет развиваться следующим образом:

Начинаем с начального множества кандидатов:

[ [a] ]

Порождаем продолжение пути [a]:

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

Удаляем первый путь из множества кандидатов и порождаем его продолжения:

[ [d,b,a], [e,b,a] ]

Добавляем список продолжений в конец списка кандидатов:

[ [c,a], [d,b,a], [e,c,a] ]

Удалим [c,a], а затем добавляем все его продолжения в конец множества кандидатов:

[ [d,b,a], [e,b,a], [f,c,a], [g,c,a] ]

После того, как пути [d,b,a] и [e,b,a] будут продолжены, измененный список кандидатов примет вид

[ [f,c,a], [g,c,a], [h,d,b,a], [i,e,b,a], [j,e,b,a] ]

В этот момент обнаруживается путь [f,c,a], содержащий целевую вершину f. Этот путь выдается в качестве решения.

Рассмотрим текст программы поиска в ширину.

**after** – состояние дуг исходного графа

**but** – целевые вершины.

**conc([],L,L).**

**conc([X|L1],L2,[X|L3]):-conc(L1,L2,L3).**

**member(X,[X|Q]).**

**member(X,[H|Q]):-member(X,Q).**

**after(a,b).**

**after(a,c).**

**after(b,d).**

**after(b,e).**

**after(c,f).**

**after(c,g).**

**after(d,h).**

**after(e,i).**

**after(e,j).**

**after(f,k).**

**goal(f).**

**goal(j).**

**solv(Start,Solv):-**

**large([[Start]],Solv).**

**large([[N|T]|\_],[N|T]):-**

**goal(N).**

**large([[B|T]|TT],Solv):-**

**bagof([B1,B|T],(after(B,B1),not(member(B1,[B|T]))),NT),**

**conc(TT,NT,TT1),!,large(TT1,Solv);**

**large(TT,Solv).**

**3.3. ЭВРИСТИЧЕСКИЙ ПОИСК: ПОИСК С ПРЕДПОЧТЕНИЕМ**

При поиске в ширину мы искали кратчайший возможный путь, т. е. путь через наименее удаленные узлы, достигнутые в ходе поиска. Эвристически поиск является расширенным вариантом поиска в ширину. При поиске по заданному критерию мы оцениваем не просто глубину пути, а у нас имеется эвристическая оценка каждого возможного узла и для продолжения пути выбирается наилучший узел в соответствии с данной оценкой.

Будем предполагать, что для графа состояний определена функция стоимости его дуг и введем функция s(a,b) равную стоимости перемещения из вершины а в вершину b. Введем эвристическую функцию f(b) равная «сложности достижения» вершины b. В соответствии с данным определением эвристической функции, из всех возможных узлов наиболее предпочтительным является тот, у которого значение эвристической функции принимает минимальное значение.

Сформируем функцию f(n) таким образом, что бы она оценивала стоимость наилучшего пути решения от начального узла до целевого, при прохождении пути через узел n. Пусть такой путь существует и t – целевая вершина, для которой этот путь минимален. Тогда оценку f(n) можно представить как сумму:

f(n)=g(n)+h(n),

где g(n) – оценка оптимального пути из s в n; h(n) – оценка пути из n в t



Вершине n будет соответствовать следующая ситуация: путь из s в n уже найден, и его стоимость может быть вычислена как сумма стоимостей составляющих его дуг. Этот путь вовсе не должен быть оптимальным, однако стоимость этого пути можно использовать в качестве оценки g(n) минимальной стоимости пути из s в n. Относительно второго слагаемого h(n) сказать что-либо трудно, поскольку пространство состояний от n до t пока еще не изучено. Поэтому о значении h(n) можно строить догадки на основании эвристических соображений, т.е. на основании знаний, которые могут быть извлечены из конкретной задачи. Заметим, что универсального метода определения функции h(n) н существует.

Процесс поиска с предпочтением состоит из некоторого числа конкурирующих подпроцессов, каждый из которых занимается своей альтернативой, т.е. просматривает свое поддерево. У деревьев есть свои поддеревья, их просматривают подпроцессы процессов и т.д. В каждый конкретный момент среди конкурирующих процессов активен только один. Очевидно таким процессом становится тот, который занят наиболее перспективной альтернативой, т.е. с минимальным значением f. Остальные процессы будут ждать до тех пор, пока f-оценки изменятся и какая-нибудь альтернатива станет наиболее перспективной. Тогда осуществляется переключение на эту альтернативу. Механизм активизации процессов функционируют так: процесс, обрабатывающий текущую альтернативу высшего приоритета, остается активным, пока функция f минимальна из всех просматриваемых альтернатив. Активный процесс углубляется вдоль дерева поиска в направлении к целевой вершине. Оценка состояния активного процесса определяется значением эвристической функции h. Это значение сравнивается со значением этой функции ближайшей альтернативы. Как только это значение текущего активного процесса превысит значение оценки соседней альтернативы, произойдет переключение на процесс, обслуживающий соседнюю альтернативу.

На рис. 1.4. приведен пример поведения конкурирующих процессов. Здесь представлена карта маршрутов перемещения из пункта s в целевой пункт t. Оценка стоимости остатка маршрута из пункта X до цели выполняется "вычислением" расстояния расст(X,t) прямой "видимости" от X до t. Таким образом формируем оценочную функцию:

f(X)=g(X)+расст(X,t).



Рис.1.4. Поиск кратчайшего маршрута из s в t.

а) Карта со связями между пунктами; связи помечены своими длинами; в квадратиках указаны расстояния по прямой до цели t.

b) Порядок, в котором при поиске с предпочтением происходит обход пунктов. Эвристические оценки основаны на расстояниях по прямой. Пунктирной линией помечено переключение активности между альтернативными путями. Эта линия задает порядок, в котором вершины принимаются для продолжения пути, а не тот порядок, в котором они порождаются.

Можно считать, что поиск в примере состоит из двух процессов. Каждый процесс прокладывает свой путь – один из двух альтернативных: I процесс проходит через а, II процесс – через е. Вначале I процесс более активен, поскольку значения f вдоль выбранного пути меньше, чем вдоль второго пути. Когда I процесс достигнет пункта с, а II процесс все еще находится в пункте е, ситуация меняется:

f(c)=g(c)+h(c)=6+4=10;

f(e)=g(e)+h(e)=2+7=9.

Поскольку f(c)>f(e), процесс 2 переходит к , а процесс 1 ждет. В пункте f имеем:

f(f)= 7+4=11;

f(c)=10,

f(c)<f(f)

поэтому II процесс предает управление I-у процессу, и движение продолжается до пункта d, так как f(d)=12>11. В этом пункте вновь активизируется процесс 2, который и завершает путь до цели t.

В программе множество деревьев-кандидатов представлены в виде дерева. Дерево изображается в программе как терм, имеющий одну из форм:

1. l(B,F/G) – дерево, состоящее из одной вершины (листа); B –вершина пространства состояний, G – g(B) (стоимость уже найденного пути из стартовой вершины в B); F – f(B)=G+h(B).

2. tr(N,F/G,TT) – дерево с непустыми поддеревьями; N – корень дерева, TT– список поддеревьев; G – g(B); F – уточненное значение f(B), т.е. значение f для наиболее перспективного преемника вершины B; список ТТ упорядочен в порядке возрастания f-оценок поддеревьев.

Уточнение значение f необходимо для того, чтобы дать программе возможность распознать наиболее перспективное поддерево (т.е. поддерево, содержащее наиболее перспективную концевую вершину) на любом уровне дерева поиска. Эта модификация f-оценок на самом деле приводит к обобщению, расширяющему область определения функции f. Теперь функция f определена не только на вершинах, но и на деревьях. Для одновершинных деревьев (листов) n остается первоначальное определение f(n)=g(n)+h(n). А для дерева Т с корнем n, имеющим m1, m2, ..., преемников получаем

.

Пояснения к программе.

|  |  |
| --- | --- |
| Р | Путь между стартовой вершиной и корнем дерева Т. |
| Т | Текущее (под)дерево поиска. |
| Extr | Предельное значение f-оценки, при котором допускается расширение. |
| Т1 | Дерево Т, расширенное в пределах ограничения Extr; f-оценка дерева Т1 больше, чем Extr (если только при расширении не была обнаружена целевая вершина. |
| Is\_solv | Индикатор, принимающий значения yes, no, never. |
| Solve | Решающий путь, ведущий из стартовой вершины через дерево Т1 к целевой вершине и имеющий стоимость, не превосходящую Extr (если такая целевая вершина обнаружена). |

Пояснения к некоторым предикатам.

**Т=tr(B,F/G,[T|TT])** – дерево Т имеет поддеревья. Расширению подвергается наиболее перспективное дерево Т. В качестве ограничения этому дереву выдается не Extr, а некоторое, возможно, меньшее значение Extr1, зависящее от f-оценок других конкурирующих поддеревьев ТТ. Так гарантируется, что растущее дерево – это всегда перспективное дерево. После расширения самого перспективного кандидата процедура (предикат) continue в зависимости от типа результата либо выдает результат, если найдено решение, либо продолжает процесс расширения.

**Т=l(B,F/G)** – порождает всех преемников вершины В вместе со стоимостями дуг, ведущих в них из В.

**suc\_list** – формирует список поддеревьев, соответствующих вершинам-преемникам, а также вычисляет их g- и f-оценки. Полученное дерево подвергается расширению с учетом ограничения Extr. Если преемников нет, то переменной Is\_solv придается значение never и в результате лист В из рассмотрения устраняется.

**аfter(B,B1,C)** – В1 – преемник вершины В; С – стоимость дуги, ведущей из В в В1.

**h(B,H)** – H – эвристическая оценка стоимости оптимального пути из вершины В в целевую вершину.

**Max\_f(Fmax)** – Fmax – некоторое значение, задаваемое пользователем. Это значение должно быть больше любой возможной f-оценки.

Рассмотрим текст программы поиска с предпочтением (эвристического поиска)

**evrpoisk(Start,Solve):-**

**max\_f(Fmax), % Fmax > любой f-оценки**

**propag([],l(Start,0/0),Fmax,\_,yes,Solve).**

**propag(P,l(B,\_),\_,\_,yes,[B|P]):-**

**goal(B). % рассматриваемый лист – цель поиска.**

**propag(P,l(B,F/G),Extr,Tree1,Is\_solv,Solve):-**

**F=<Extr, % получение дерева из приемников листа**

**bagof(B1/C,(after(B,B1,C),not(member(B1,P))),Successers),**

**!,suc\_list(G,Successers,TT),**

**opt\_f(TT,F1),**

**propag(P,tr(B,F1/G,TT),Extr,Tree1,Is\_solv,Solve).**

**propag(P,l(B,F/G),Extr,Tree1,never,Solve):-**

**F=<Extr. % Нет приемников – тупик**

**propag(P,tr(B,F/G,[T|TT]),Extr,Tree1,Is\_solv,Solve):-**

**F=<Extr, % Продолжить дерево**

**opt\_f(TT,OF),**

**min(Extr,OF,Extr1),**

**propag([B|P],T,Extr1,T1,Is\_solv1,Solve),**

**continue(P,tr(B,F/G,[T1,TT]),Extr,Tree1,Is\_solv1,Is\_solv,Solve).**

**propag(\_,tr(\_,\_,[]),\_,\_,never,\_):-!. % Тупиковое дерево - нет решений**

**propag(\_,Tree,Extr,Tree,no,\_):-**

**f(Tree,F),F>Extr. % Рост остановлен**

**continue(\_,\_,\_,\_,yes,yes,Solve).**

**continue(P,tr(B,F/G,[T1,TT]),Extr,Tree1,no,Is\_solv,Solve):-**

**insert(T1,TT,NTT),**

**opt\_f(NTT,F1),**

**propag(P,tr(B,F1/G,NTT),Extr,Tree1,Is\_solv,Solve).**

**continue(P,tr(B,F/G,[T1,TT]),Extr,Tree1,never,Is\_solv,Solve):-**

**opt\_f(TT,F1),**

**propag(P,tr(B,F1/G,TT),Extr,Tree1,Is\_solv,Solve).**

**suc\_list(\_,[],[]).**

**suc\_list(G0,[B/C|BB],TT):-**

**G is G0+C,**

**h(B,H), % Эвристика h(B)**

**F is G+H,**

**suc\_list(G0,BB,TT1),**

**insert(l(B,F/G),TT1,TT).**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**\*Вставка дерева Т в список деревьев ТТ с сохранением \***

**\* упорядоченности по f-оценкам \***

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**insert(T,TT,[T|TT]):-**

**f(T,F),**

**opt\_f(TT,F1),**

**F=<F1,!.**

**insert(T,[T1|TT],[T1|TT1]):-**

**insert(T,TT,TT1).**

**/\*\*\*\*\*\*Проверка принадлежности элемента списку\*\*\*\*\*\*/**

**member(X,[X|Q]).**

**member(X,[H|Q]):-member(X,Q).**

**/\*\*\*\*\*\*\* Получение f-оценки \*\*\*\*\*\*\*\*\*/**

**f(l(\_,F/\_),F). % f-оценка листа**

**f(tr(\_,F/\_,\_),F). % f-оценка дерева**

**opt\_f([T|\_],F):- % наилучшая f-оценка для**

**f(T,F). % списка деревьев**

**opt\_f([],Fmax):- % нет деревьев:**

**max\_f(Fmax). % плохая f-оценка**

**min(X,Y,X):-**

**X=<Y,!.**

**min(X,Y,Y).**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**\* База фактов тестового примера: \***

**\* состояние дуг исходного графа after; \***

**\* функция h оценки удаленности вершин от целевой вершины;\***

**\* целевая вершина – goal; \***

**\* максимальное значение max\_f(Fmax) оценки функции f \***

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**after(s,a,2). % База фактов:**

**after(s,e,2). % Исходный граф**

**after(a,b,2).**

**after(b,c,2).**

**after(c,d,3).**

**after(e,f,5).**

**after(f,g,2).**

**after(d,t,3).**

**after(g,t,2).**

**after(b,v,1). %Тупиковая вершина**

**goal(t). % Целевая вершина**

**h(a,5). % Значение эвристической**

**h(e,7). % функции h для оценки вершин**

**h(b,4).**

**h(f,4).**

**h(t,0).**

**h(d,3).**

**h(g,2).**

**h(c,4).**

**h(v,1).**

**max\_f(20). % Макимальное значение f-оценки**

**% Пример запроса к программе**

**%?-evripoisk(s,Path).**

**Path=[t,g,f,e,s] ->;**

**Path=[t,d,c,b,a,s] ->;**

**false**

***ЗАДАНИЕ 3.1***

Проследите трассировку для указанных в данном разделе программ.

***ЗАДАНИЕ 3.2***

Написать программу поиска пути минимального веса, в графе, ребрам которого приписаны веса, соединяющий заданную пару вершин. Вес пути равен сумме весов ребер, входящих в этот путь.

### 4.ПРОСТРАНСТВО СОСТОЯНИЙ.

***ЦЕЛЬ***. Знакомство с одной общей схемой для представления задач, называемой пространством состояний.

**4.1 ЗАДАЧА О ТРЕХ СОСУДАХ**

Рассмотрим задачу о трех сосудах. Имеется три сосуда емкостью 8, 5 и 3 литра. В первом сосуде содержится 8 литров вина. Требуется разделить вино поровну, пользуясь только этими тремя сосудами, т.е. алгоритм переливания.

Информацию об объемах сосудов представим фактом:

объемы(8,5,3).

Правило переливания из одного сосуда в другой:

Чтобы перелить из одного сосуда в другой, надо, чтобы в сосуде "откуда" было какое-то количество вина, а в сосуде "куда"- свободное место. Если в сосуде "откуда" вина больше чем

свободного места в сосуде "куда", то переливается количество, равное свободному объему сосуда "куда", в противном случае из сосуда "откуда" выливается все вино, т.е. переливаемое количество равно минимуму из количества вина в сосуде "откуда" и свободного места в сосуде "куда". После переливания вина в сосуде "откуда" станет меньше на перелитое количество, а в сосуде "куда" увеличится на столько же. Количество вина в оставшемся третьем сосуде не

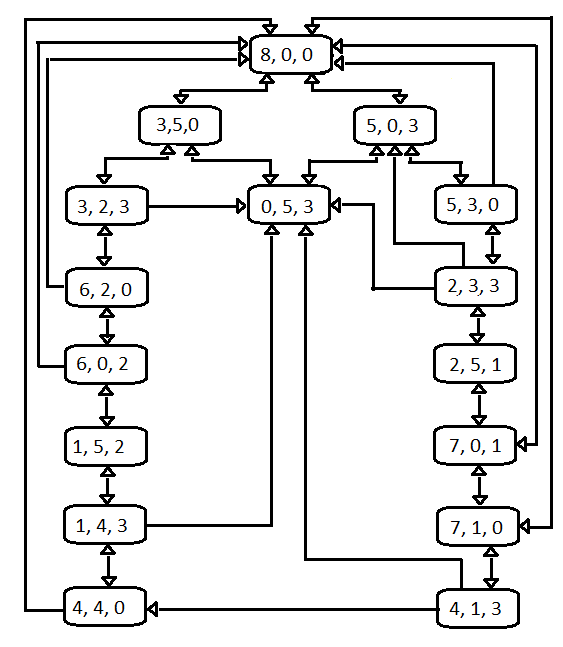
изменится. Затем надо повторить переливание из нового состояния так, чтобы после ряда переливаний получить в двух первых сосудах по 4 литра.

В этой задаче ситуацию можно описать следующим образом

состояние(Количество\_вина\_в\_первом\_сосуде, Количество\_вина\_во\_втором\_сосуде,

Количество\_вина\_в\_третьем\_сосуде).

дуги пространства состояний определяются разрешенным переходам из одной ситуации в другую, т.е. правилами переливаниями. Задача отыскания решения эквивалентна задаче построения пути между начальной ситуацией (8,0,0) и конечной ситуацией (4,0,0).



Чтобы не "зацикливаться" будем хранить все пройденные состояния в списке в виде структур сост(В\_первом,Во\_втором,В\_третьем) и проверять каждое новое состояние на принадлежность этому списку.

Для трех сосудов возможно шесть вариантов переливаний и соответственно 6 правил (1→2, 1→3, 2→1, 2→3, 3→1, 3→2) + правило остановки.

**объемы(8,5,3).**

**решение:-перелить(8,0,0,[]).**

**перелить(4,4,0,\_пройденные):-write(\_пройденные).**

**минимум(\_x,\_y,\_x):-\_x=<\_y,!.**

**минимум(\_x,\_y,\_y).**

**принадлежит(\_x,[\_x|\_]):-!.**

**принадлежит(\_x,[\_y|\_l]):-!,принадлежит(\_x,\_l),!.**

**%1->2**

**перелить(\_в\_первом,\_во\_втором,\_в\_третьем, \_пройденные):-**

**объемы(\_объем1,\_объем2,\_объем3),**

**\_в\_первом>0,**

**\_свободно is \_объем2-\_во\_втором,**

**\_свободно>0,**

**минимум(\_в\_первом,\_свободно, \_количество),**

**\_стало\_в\_первом is \_в\_первом-\_количество,**

**\_стало\_во\_втором is \_во\_втором+\_количество,**

**\_новое=сост(\_стало\_в\_первом,\_стало\_во\_втором,\_в\_третьем),**

**not(принадлежит(\_новое,\_пройденные)),**

**перелить(\_стало\_в\_первом,\_стало\_во\_втором,\_в\_третьем,[\_новое|\_пройденные]).**

***ЗАДАНИЕ 4.1***

Допишите оставшиеся правила допустимого переливания. Проверьте правильность работы программы, задав вопрос ?-решение.

Другой вариант решения той же задачи:

**объемы(8,5,3).**

**исходное(состояние(8,0,0)).**

**цель(состояние(4,4,0)).**

**решение:-исходное(\_s),перелить([\_s]).**

**перелить([\_s|\_p]):- цель(\_s),отобразить\_решение([\_s|\_p]).**

**перелить([состояние(\_1,\_2,\_3)|\_p]):-**

**допустимо(\_1,\_2,\_3,\_n1,\_n2,\_n3),g**

**not(принадлежит(состояние(\_n1,\_n2,\_n3),**

**[состояние(\_1,\_2,\_3)|\_p])),**

**перелить([состояние(\_n1,\_n2,\_n3),**

**состояние(\_1,\_2,\_3)|\_p]).**

**перелито(\_1,\_2,\_f,\_n1,\_n2):- минимум(\_1,\_f,\_k), \_n1 is \_1 - \_k,\_n2 is \_2 + \_k.**

**минимум(\_x,\_y,\_x):-\_x=<\_y,!.**

**минимум(\_x,\_y,\_y).**

**принадлежит(\_x,[\_x|\_]):-!.**

**принадлежит(\_x,[\_y|\_l]):-!,принадлежит(\_x,\_l),!.**

**отобразить\_решение([]):-!.**

**отобразить\_решение([\_x|\_l]):-отобразить\_решение(\_l),!,write(\_x),nl.**

**%1->2**

**допустимо(\_1,\_2,\_3,\_n1,\_n2,\_3):-**

**\_1>0,объемы(\_v1,\_v2,\_v3),**

**\_f is \_v2-\_2,**

**\_f>0,**

**перелито(\_1,\_2,\_f,\_n1,\_n2).**

***ЗАДАНИЕ 4.2***

Допишите оставшиеся 5 правил допустимого переливания и проверьте правильность работы программы.

# Применение поиска по заданному критерию для решения головоломки "игра в восемь"

Рассмотрим головоломку «игра в восемь» Головоломка состоит из восьми скользящих фишек, пронумерованных цифрами от 1 до 8 и размещенных в коробке с размерами 3\*3, с девятью ячейками. Одна из ячеек всегда пуста, и в эту пустую ячейку может быть передвинута по горизонтали или по вертикали любая соседняя фишка. Это правило можно выразить иначе: пустую ячейку можно перемещать по коробке, меняя ее местами с любой из соседних фишек. Конечной ситуацией является некоторое особое расположение фишек, как показано, например, на рис. 5.1. И нам необходимо найти путь с наименьшим числом перемещений от начальной ситуации до целевой.

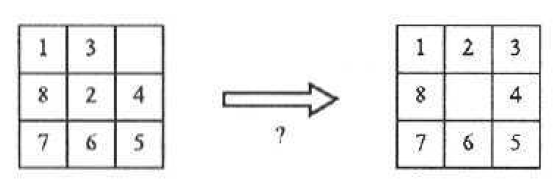


рис 5.1

Так как нам надо найти путь с минимальной длиной, необходимо использовать эвристический поиск. Чтобы можно было применить программу поиска по заданному критерию, для решения некоторой конкретной задачи, необходимо определить отношения, характеризующие рассматриваемую проблему. Эти отношения характеризуют конкретную проблему ("правила игры"), а также воплощают в себе эвристическую информацию о том, как решить данную задачу. Такая эвристическая информация предоставляется в форме эвристической функции.

Предикаты, касающиеся конкретной проблемы, задаются в виде следующего отношения:

s( Node, Nodel, Cost)

Это отношение является истинным, если в пространстве состояний между узлами Mcde и Nodel имеется дуга со стоимостью Cost. Отношение

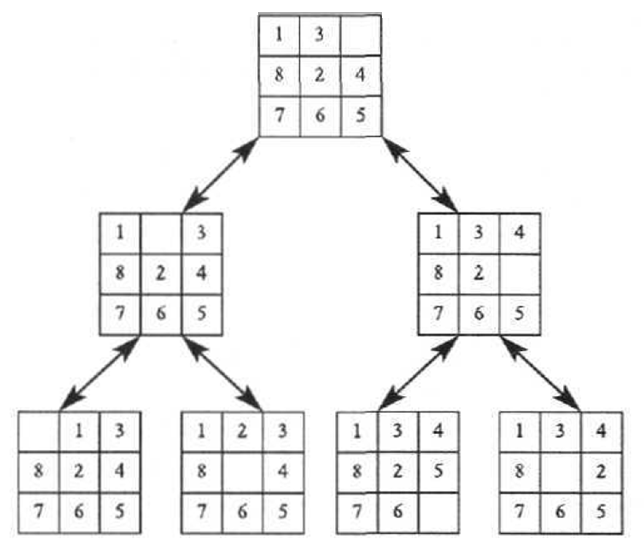
goal( Mode)

является истинным, если Node — целевой узел в пространстве состояний. А в отношении

h( Node, H)

переменная Н — эвристическая оценка стоимости пути с минимальной стоимостью от узла Node до целевого узла.

На рис. 5.2 рассматривает головоломка «игра в восемь» и ее представление в виде проблемы поиска пути.

Рис. 5.2

Любой из узлов в пространстве состояний представляет собой некоторую конфигурацию фишек в коробке. В данной программе эта конфигурация представлена в виде списка текущих положений фишек. Каждая позиция обозначается парой координат, X/Y. Порядок элементов в списке является следующим.

1. Текущая позиция пустого квадрата.

2. Текущая позиция фишки 1.

3. Текущая позиция фишки 2.

…

Целевая ситуация (см. рис. 5.1), в которой все фишки находятся на своих исходных клетках, определяется следующим предложением:

goal! [2/2,1/3,2/3,3/3,3/2,3/1,2/1,1/1,1/2]).

Принято считать, что пустая клетка - это "пустая" фишка, которая может передвигаться по горизонтали или г.о вертикали на место, занимаемое любой из ее соседних фишек; это означает, что "пустая" фишка и ее соседняя фишка могут поменяться местами. Рассмотрим текст программы решения задачи «игра в восемь».

**s([Empty|Tiles],[Tile|Tiles1],1):-swap( Empty, Tile, Tiles, Tiles1).**

**% Стоимости всех дуг равны 1**

**% Поменять местами пустую фишку**

**% Empty и фишку Tile в списке Tiles**

**swap( Empty, Tile, [Tile | Ts], [Empty | Ts] ) :-**

**mandist( Empty, Tile, 1). % Манхэттенское расстояние равно 1**

**swap( Empty, Tile, [T1 | Ts], [T1 | Ts1] ) :-**

**swap( Empty, Tile, Ts, Ts1).**

**mandist( X/Y, X1/Y1, D) :- dif( X, X1, Dx),**

**dif( Y, Y1 , Dy) ,**

**D is Dx + Dy. % D - это манхэттенское расстояние**

**% между двумя клетками**

**dif( A, B, D) :- % D равно IA-B|**

**D is A-B, D >= 0,!;**

**D is B-A.**

**% Эвристическая оценка h представляет собой сумму расстояний от каждой фишки**

**% до ее "исходной" клетки плис утроенное значение "оценки упорядоченности"**

**h([Empty | Tiles], H) :-**

**goal( [Empty1 | GoalSquares] ),**

**totdist( Tiles, GoalSquares, D), % Суммарное расстояние от исходных клеток**

**seq( Tiles, S), % Оценка упорядоченности**

**H is D + 3\*S.**

**totdist( [], [], 0) .**

**totdist( [Tile | Tiles], [Square | Squares], D) :-**

**mandist( Tile, Square, D1),**

**totdist( Tiles, Squares, D2) ,**

**D is D1 + D2.**

**% seq( TilePositions, Score): оценка упорядоченности**

**seq( [First | OtherTiles], S) :-**

**seq( [First | OtherTiles], First, S).**

**seq( [Tile1, Tile2 | Tiles], First, S) :-**

**score( Tile1, Tile2, S1),**

**seq( [Tile2 | Tiles], First, S2),**

**S is S1 + S2.**

**seq( [Last], First, S) :-**

**score( Last, First, S).**

**score( 2/2, \_, 1):-!. % Оценка фишки, стоящей в центре, равна 1**

**score( 1/3, 2/3,0):- !. % Оценка фишки, за которой следует**

**% допустимый преемник, равна 0**

**score( 2/3, 3/3, 0) :- ! .**

**score( 3/3, 3/2, 0) :- !.**

**score( 3/2, 3/1, 0) :- !.**

**score( 3/1, 2/1, 0) :- !.**

**score( 2/1, 1/1, 0) :- !.**

**score( 1/1, 1/2, 0) :- !.**

**score( 1/2, 1/3, 0) :- !.**

**score( \_, \_, 2). % Оценка фишки, за которой следует**

**% недопустимый преемник, равна 2**

**goal( [2/2,1/3,2/3,3/3,3/2,3/1,2/1,1/1,1/2] ). % Исходные клетки для фишек**

**% Показать путь решения как список позиций на доске**

**showsol( [] ).**

**showsol( [P | L] ) :-**

**showsol( L),**

**nl, write('---'),**

**showpos( P).**

**% Показать позицию на доске**

**showpos( [S0,S1,S2,S3,S4,S5,S6,S7,S8] ) :-**

**member( Y, [3,2,1] ), % Последовательность координат Y**

**nl, member( X, [1,2,3] ), % Последовательность координат х**

**member( Tile-X/Y, % Фишка в клетке X/Y**

**[' '-S0,1-S1,2-S2,3-S3,4-S4,5-S5, 6-S6,7-S7, 8-S8 ] ),**

**write( Tile),**

**fail; % Выполнить перебор с возвратом к следующей клетке**

**true. % Обработка всех клеток закончена**

**% Начальные позиции для некоторых задач**

**start1([2/2,1/3,3/2,2/3,3/3,3/1,2/1,1/1,1/2] ). % Требует 4 хода**

**start2( [2/1,1/2,1/3,3/3,3/2,3/1,2/2,1/1,2/3] ). % Требует 5 ходов**

**start3( [2/2,2/3,1/3,3/1,1/2,2/1,3/3,1/1,3/2] ). % Требует 19 ходов**

**% Пример запроса: ?- start1( Pos), bestfirst(Pos, Sol), showsol(Sol).**

**%-----Поиск по заданному критерию**

**evrpoisk(Start,Solve):-**

**max\_f(Fmax), % Fmax > любой f-оценки**

**propag([],l(Start,0/0),Fmax,\_,yes,Solve).**

**propag(P,l(B,\_),\_,\_,yes,[B|P]):-**

**goal(B). % рассматриваемый лист – цель поиска.**

**propag(P,l(B,F/G),Extr,Tree1,Is\_solv,Solve):-**

**F=<Extr, % получение дерева из приемников листа**

**bagof(B1/C,(s(B,B1,C),not(member(B1,P))),Successers),**

**!,suc\_list(G,Successers,TT), %after -s**

**opt\_f(TT,F1),**

**propag(P,tr(B,F1/G,TT),Extr,Tree1,Is\_solv,Solve).**

**propag(P,l(B,F/G),Extr,Tree1,never,Solve):-**

**F=<Extr. % Нет приемников – тупик**

**propag(P,tr(B,F/G,[T|TT]),Extr,Tree1,Is\_solv,Solve):-**

**F=<Extr, % Продолжить дерево**

**opt\_f(TT,OF),**

**min(Extr,OF,Extr1),**

**propag([B|P],T,Extr1,T1,Is\_solv1,Solve),**

**continue(P,tr(B,F/G,[T1,TT]),Extr,Tree1,Is\_solv1,Is\_solv,Solve).**

**propag(\_,tr(\_,\_,[]),\_,\_,never,\_):-!. % Тупиковое дерево - нет решений**

**propag(\_,Tree,Extr,Tree,no,\_):-**

**f(Tree,F),F>Extr. % Рост остановлен**

**continue(\_,\_,\_,\_,yes,yes,Solve).**

**continue(P,tr(B,F/G,[T1,TT]),Extr,Tree1,no,Is\_solv,Solve):-**

**insert(T1,TT,NTT),**

**opt\_f(NTT,F1),**

**propag(P,tr(B,F1/G,NTT),Extr,Tree1,Is\_solv,Solve).**

**continue(P,tr(B,F/G,[T1,TT]),Extr,Tree1,never,Is\_solv,Solve):-**

**opt\_f(TT,F1),**

**propag(P,tr(B,F1/G,TT),Extr,Tree1,Is\_solv,Solve).**

**suc\_list(\_,[],[]).**

**suc\_list(G0,[B/C|BB],TT):-**

**G is G0+C,**

**h(B,H), % Эвристика h(B)**

**F is G+H,**

**suc\_list(G0,BB,TT1),**

**insert(l(B,F/G),TT1,TT).**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**\*Вставка дерева Т в список деревьев ТТ с сохранением \***

**\* упорядоченности по f-оценкам \***

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**insert(T,TT,[T|TT]):-**

**f(T,F),**

**opt\_f(TT,F1),**

**F=<F1,!.**

**insert(T,[T1|TT],[T1|TT1]):-**

**insert(T,TT,TT1).**

**/\*\*\*\*\*\*Проверка принадлежности элемента списку\*\*\*\*\*\*/**

**member(X,[X|Q]).**

**member(X,[H|Q]):-member(X,Q).**

**/\*\*\*\*\*\*\* Получение f-оценки \*\*\*\*\*\*\*\*\*/**

**f(l(\_,F/\_),F). % f-оценка листа**

**f(tr(\_,F/\_,\_),F). % f-оценка дерева**

**opt\_f([T|\_],F):- % наилучшая f-оценка для**

**f(T,F). % списка деревьев**

**opt\_f([],Fmax):- % нет деревьев:**

**max\_f(Fmax). % плохая f-оценка**

**min(X,Y,X):-**

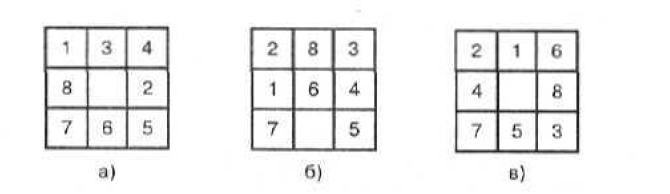
**X=<Y,!.**

**min(X,Y,Y).**

**max\_f(9999). % Макимальное значение f-оценки**

При определении этих отношений применяется следующее вспомогательное отношение:

**mandist ( SI, S2, D).**

где D — *манхэттенское расстояние* между клетками S1 и S2; оно измеряется как сумма расстояний между S1 и S2 в горизонтальном и вертикальном направлениях. Необходимо найти путь к решению минимальной длины. Поэтому определим стоимость всех дуг в пространстве состояний как равную 1. В программе, заданы также три примера начальных позиций, которые составлены по диаграммам, показанным на рис. 5.3.*Рис. 5.3 Три начальные позиции для головоломки "игра е восемь":*

*а) требует. 4 хода; б) требует. 5 ходов; в) требует 18 ходов*

Эвристическая функция h определяется в программе следующим образом:

**h( Роs, H)**

где Pos — позиция на доске, Н — сочетание двух описанных ниже критериев;

1. totdist - *суммарное расстояние* восьми фишек в позиции Pos от клеток, в которых должны находиться фишки в целевом состоянии. Например, в начальной позиции головоломки, приведенной на рисунке а), totdist = 4.

2. seq - *оценка упорядоченности,* определяющая ту степень, в какой фишки, стоящие в текущей позиции, уже упорядочены по отношению к той последовательности, которая должна быть достигнута в целевой конфигурации. Значение seq вычисляется как сумма оценок для каждой фишки в соответствии со следующими правилами:

• фишка в центре имеет оценку 1;

• фишка, стоящая на нецентральной клетке, имеет оценку 0, если за ней в направлении по часовой стрелке следует соответствующий ей преемник (например, если за фишкой 1 следует фишка 2);

•если за фишкой не следует соответствующий ей преемник, то такая фишка имеет оценку 2.

Например, для начальной позиции головоломки, приведенной на рис.5.3 *а)* seq = 6.

Эвристическая оценка, Н, вычисляется как H = totdist + 3 \* seq.

Эта эвристическая функция действует успешно в том смысле, что очень эффективно направляет поиск к цели. Например, при решении задач, показанных на рис. 5.3 *а), б),* не происходило даже развертывание ни одного узла, выходящего за пределы кратчайшего пути решения, до того, как было найдено первое решение. Это означает, что в таких случаях кратчайшие пути решения отыскиваются непосредственно, без какого-либо перебора с возвратами. Почти сразу же решается даже трудная задача, приведенная на рис.5.3 *в).* Но недостаток этой эвристики состоит в том, что она не является допустимой: она не гарантирует, что кратчайший путь решения всегда будет найден до обнаружения какого-либо более длинного решения. Дело в том, что данная функция h не удовлетворяет условию допустимости, при котором h < h\* для всех узлов. Например, для начальной позиции, приведенной на рис. 5.3 а), имеет место следующее:

h = <4 + 3\*6 = 2 2 , h\* = 4

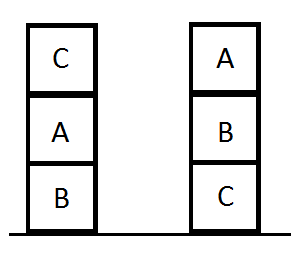
С другой стороны, допустимым является отдельно взятый критерий "суммарного расстояния", поскольку для всех позиций имеет место следующее:

totdist < *h\**

Допустимость этого отношения можно легко проверить с помощью таких рассуждений: если условия этой головоломки будут упрощены таким образом, что фишки можно будет переносить друг над другом, то каждая фишка сможет переходить к своей исходной клетке по траектории, длина которой точно равна манхэттенскому расстоянию между начальной клеткой фишки и ее исходной клеткой. Поэтому оптимальное решение в упрощенной головоломке будет точно равно длине totdist . Но в первоначальном варианте задачи фишки могут мешать друг другу и одна из них может находиться на пути другой, что препятствует движению фишек по кратчайшим траекториям, а это гарантирует, что длина оптимального решения равна или больше чем totdist .

***ЗАДАНИЕ 4.3***

Построить граф пространства состояний для задачи переупорядочевания кубиков, поставленных друг на друга, как показано на рисунке.



Начальная Конечная

ситуация ситуация

На каждом шагу разрешается переставлять только один кубик. Кубик можно взять только, если он лежит сверху. Кубик можно поставить либо на стол, либо на другой кубик. Напишите программу переупорядочивания кубиков.

***ЗАДАНИЕ 4.5***

Решите задачу о переливании для сосудов емкостью 10,7 и 3 л.

***ЗАДАНИЕ 4.6***

("Волк, коза и капуста"). Перевозчику нужно переправить через реку волка, козу и мешок с капустой. Лодка так мала, что кроме перевозчика может взять только один из этих объектов. Кроме того, капусту нельзя оставлять вместе с козой, а козу с волком. Как можно осуществить переправу?

Напишите программу для решения задачи "Волк, коза и капуста".

***ЗАДАНИЕ 4.7***

Решите одну из следующих задач, согласно порядковому номеру в группе:

1.(“Миссионеры и каннибалы”). Три миссионера и три каннибала находятся на левом берегу реки. Здесь же - лодка, вмещающая не более двух человек. Все хотят перебраться на другой берег. Если на каком-либо берегу каннибалов окажется больше, чем миссионеров, то каннибалы съедят оставшихся в меньшинстве миссионеров. Найти последовательность ездок, гарантирующих безопасную переправу на правый берег. Напишите программу для решения задачи "Миссионеры и каннибалы".

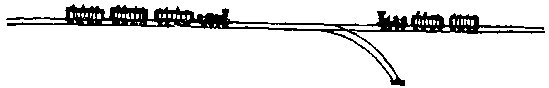
2.  Решите задачу о "Миссионерах и каннибалах" для случая, когда только один миссионер и каннибал умеют грести.

3.(“Ревнивые мужья”). Три ревнивых мужа и их жены должны переправиться через реку. имеется только одна маленькая лодка, которая может выдержать одновременно только двоих. Как могут переправиться все шестеро, если никакой муж не оставит жену в присутствии других мужчин. Напишите программу для решения задачи "Ревнивые мужья".

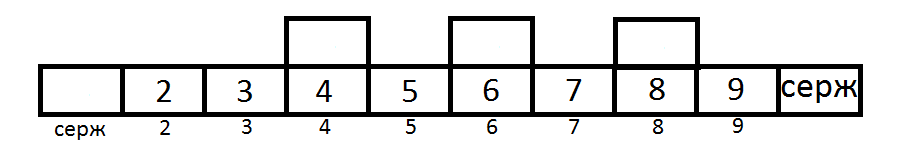
4.  Модифицируйте программу для случая пяти супружеских пар и лодки, вмещающей 3 человека.

5. (“Железнодорожная стрелка”). Два поезда с n и m вагонами смогут разминуться с помощью изображенной здесь стрелки и продолжить движение дальше вперед паровозами. Небольшой боковой тупик достаточен для того, чтобы принять либо один паровоз, либо

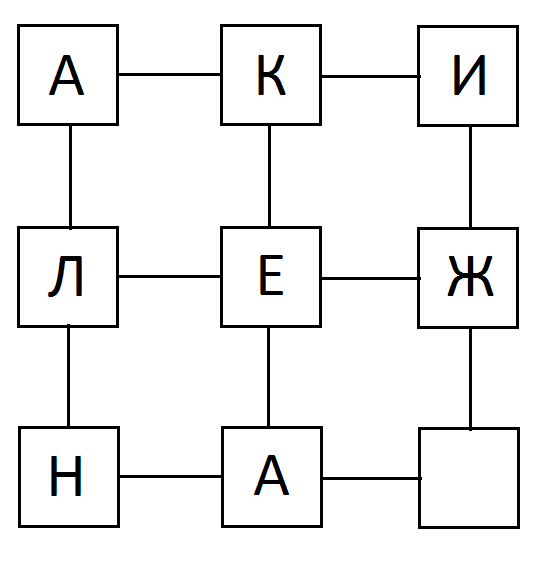
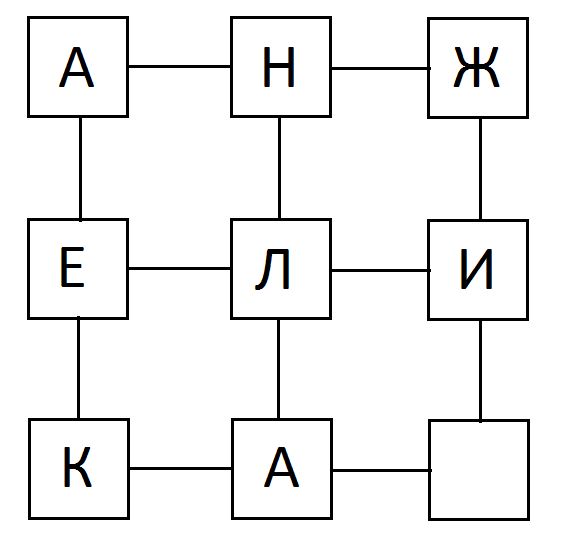
один вагон.



6. (“Солдаты в окопе”). На рисунке изображены 8 солдат и сержант в окопе. Сержант хочет перебраться на другой конец окопа, но так чтобы при этом все остальные солдаты остались на своих местах. Окоп слишком узок и вдвоем в нем не разойтись. (На рисунке в клеточках изображено начальное положение солдат и сержанта в окопе, а под клеточками – конечное.)



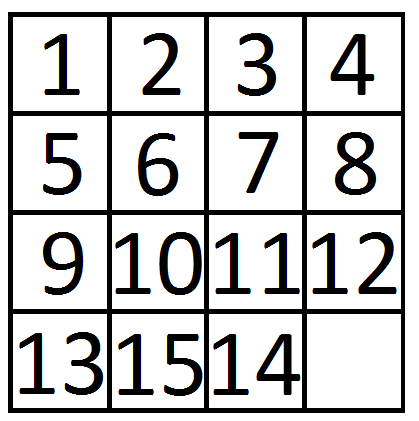
7. (“Анжелика”).

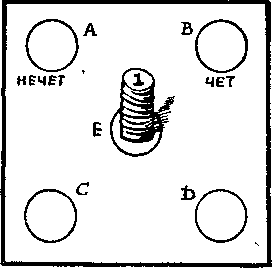
Передвигая фишки вдоль прямых на свободные места составить слово АНЖЕЛИКА.

8. (“Поссорившиеся пары”). Три супружеские пары должны перебраться через реку в маленькой лодке, которая может выдержать одновременно только двоих. Мистер С поссорился с двумя другими джентельменами, а миссис С перестала разговаривать с остальными леди. Как могут переправиться все шестеро, если ни одна из женщин не умеет грести и никакие два человека, находящихся в ссоре, не переправляются одновременно и даже не находятся одновременно на одном берегу.

9. (“Игра в пятнадцать”) На рисунке перед Вами знаменитая головоломка - игра в 15, в которой требовалось, передвигая фишки в коробке, расположить 14 и 15 в правильном порядке.

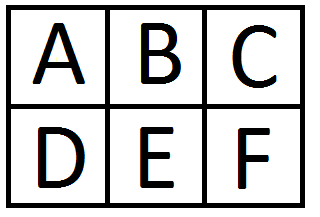


10. (“Четные и нечетные фишки”). Стопка из 8 фишек помещена в центральный квадрат, как показано на рисунке.



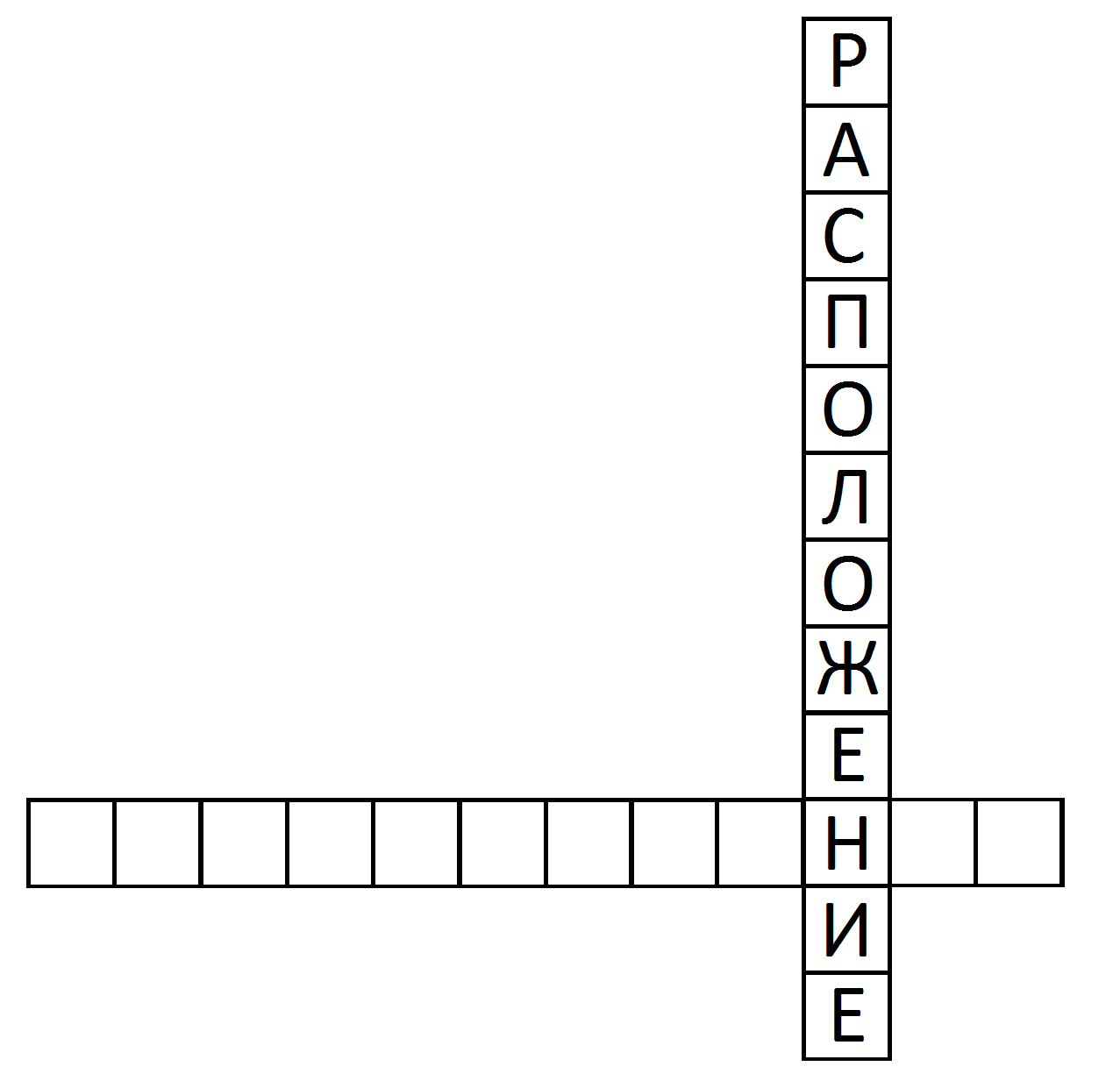
Фишки перенумерованы, таким образом, чтобы сверху вниз номера шли по порядку от 1 до 8. Требуется поместить фишки 1, 3, 5, 7 в квадрат с надписью НЕЧЕТ, а 2, 4, 6, 8 - в квадрат с надписью ЧЕТ. За один раз разрешается перемещать из квадрата в квадрат лишь одну фишку, причем больший номер нельзя класть на меньший, запрещается также помещать фишки с номерами разной четности одновременно в один и тот же квадрат.

11. (“Перемещение фишек”). Игральная доска разделена на 6 квадратов, как показано на рисунке.

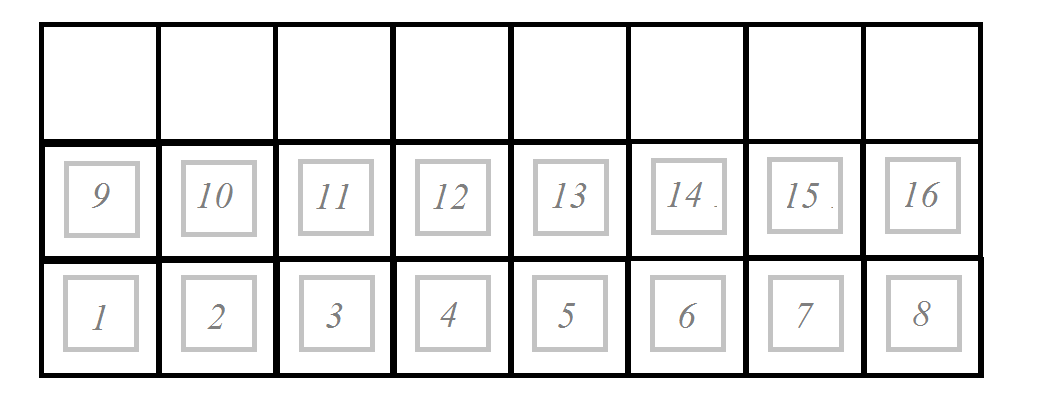


В квадрате A помещена стопка из 15 фишек с номерами 1, 2, 3,..., 15, идущими сверху вниз. Надо поместить всю стопку в квадрат F. Перемещать можно по одной фишке за ход в любой квадрат, но больший номер нельзя класть на меньший.

12. (“Китайская головоломка”).Измените расположение слова РАСПОЛОЖЕНИЕ, буквы могут передвигаться только по двум желобам.



13. ("Упрощенный солитер") На игральной доске помещены 16 пронумерованных фишек, как показано на рисунке.



Одна фишка может перепрыгнуть через другую на свободный квадрат, расположенный за этой последней, причем та, через которую перепрыгнули, удаляется. Совершая последовательные прыжки, надо удалить все фишки, кроме одной. Движение по диагонали запрещено.