Министерство образования и науки Российской Федерации

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

С. Г. ПОПОВ

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ ПРОЛОГ

Лабораторный практикум

Санкт-Петербург
2014

УДК 004.047 (076.5)

ПОПОВ С.Г. Программирование на языке ПРОЛОГ: Лабораторный практикум. 2014. 49 с.

Практикум содержит справочные материалы и примеры программ для реализации рекурсивно-логических алгоритмов в среде программирования Turbo-Prolog, и руководство по выполнению лабораторных работ. Для каждой лабораторной работы приведены постановка задачи, алгоритм на логическом псевдокоде, пример исходного текста программы, анализ сложности предложенного алгоритма и тестовые примеры.
Практикум предназначен для студентов направлений подготовки бакалавров «Математика и компьютерные науки» и «Информатика и вычислительная техника».
Иллюстраций 20.

© Попов С. Г.

Содержание

1. Лабораторная работа 1. Вычисление факториала числа и чисел Фибоначчи. 4

2. Лабораторная работа 2. Управление списками. 7

3. Лабораторная работа 3. Задача о расстановке 8 ферзей. 13

4. Лабораторная работа 4. Задача об обходе шахматного поля конем. 16

5. Лабораторная работа 5. Сортировки списков. 17

6. Лабораторная работа 6. Управление двоичными деревьями поиска. 20

7. Лабораторная работа 7. Операции на графе. 24

8. Лабораторная работа 8. Построение остовных деревьев графа. 28

9. Решение математической головоломки. 32

10. Лабораторная работа 9. Задача о волке козе и капусте. 35

11. Лабораторная работа 10. Задача о ханойской башне. 39

12. Лабораторная работа 11. Задача оптимального распределения ресурсов вычислительной системы реального времени. 43

13. Литература 49

1. Лабораторная работа 1. Вычисление факториала числа и чисел Фибоначчи.

Постановка задачи. Реализовать программу на ПРОЛОГЕ, вычисляющую факториал числа и числа Фиббоначи двумя способами: при помощи простой и хвостовой рекурсий. Привести трассировку программ и сделать вывод об эффективности реализации алгоритмов.

Вычисление факториала числа. Факториалом называют произведение всех натуральных чисел от единицы до n. Факториал обозначается:

$$n!:\ n!{=}1{\times}2{\times}3{\times}\ldots{\times}n.$$

Например: $4! = 1{\times}2{\times}3{\times}4{=}24$.

Описание алгоритма. Рассмотрим рекуррентное соотношение

$$n!{=}n{\times}(n{-}1).$$

Определим отношение $factorial(N, X)$, которое означает $N!{=}X$. Введём следующие правила:

1. если $n{=}0$, то $n!{=}1$, так как $0!{=}1$;
2. если же $n{>}0$, то найдём сначала рекурсивно $(n{-}1)!$ и умножим полученное значение на n.

Код программы вычисления факториала числа.

```
1  PREDICATES
2      factorial (integer, integer)
3  CLAUSES
4      factorial (0, 1).
5      factorial (N, X) :- N > 1, N1 = N - 1,
6      factorial (N1, X1), X = X1 * N.
7  GOAL
8      factorial(5,F), write(F).
```

Результат: $F{=}120$.

Пример выполнения программы для вычисления 4! приведён на рисунке 1.

Оценка вычислительной сложности алгоритма вычисления факториала числа. Максимальная глубина рекурсии: $n{+}1$. Общее число рекурсивных вызовов: $n{+}1$. Объем памяти при максимальной глубине рекурсии: $4{\times}(sizeof(int)){\times}n$.

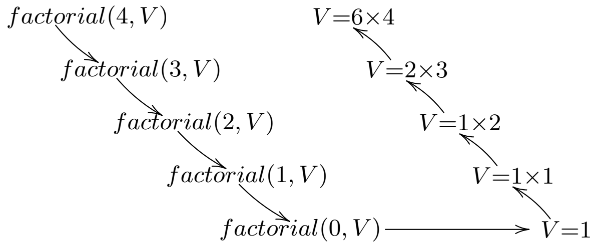

Рисунок 1. Рекурсивные вызовы правила вычисления факториала числа.

Вычисление чисел Фибоначчи. Числами Фибоначчи называются числа F_n, образующие последовательность:

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, \dots,$$

причём каждое следующее число последовательности Фибоначчи, начиная с первого, вычисляется сложением двух предыдущих:

$$F(n)=F(n-1)+F(n-2).$$

Например: $1+1=2; 1+2=3; 2+3=5; 3+5=8; 5+8=13$.

Описание алгоритма простой рекурсии. Определим отношение: $fib(N, V)$, где N — номер числа Фибоначчи по порядку, а V — результат.

Определим правила вычисления очередного числа:

1. если $N=0$ или $N=1$, то $V=1$;
2. иначе рекурсивно найдём значения чисел Фибоначчи $N-1$ и $N-2$, их сумма как раз и будет N-ым числом Фибоначчи.

Код программы рекурсивного вычисления чисел Фиббоначи.

```
1   PREDICATES
2       fib(integer, integer)
3   CLAUSES
4       fib (0, 1).
5       fib (1, 1).
6       fib (N, V) :- N > 1, N1 = N - 1, N2 = N - 2,
7                     fib (N1, V1),
8                     fib (N2, V2),
9                     V = V1 + V2.
10  GOAL
11      fib(5, F), write(F).
```

Результат: $F = 5$.

На рисунке 2 приведён пример вычисления чисел Фибоначчи последовательными рекурсивными вызовами.

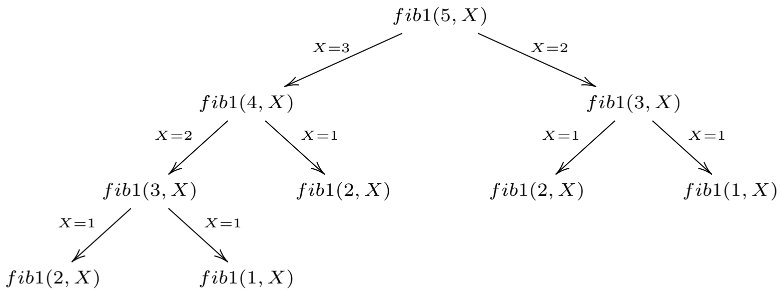

Рисунок 2. Рекурсивные вызовы правила вычисления чисел Фиббоначи.

Алгоритм вычисления чисел Фибоначчи при помощи хвостовой рекурсии. Недостатком предыдущего метода является многократное повторение вычислений $(n-2)$-го числа. Чтобы не вычислять на каждом шаге его значения, можно сохранять однажды вычисленное значение и передавать его в функцию.

Введем отношение: $fib2(N, V1, V)$, где N — номер числа, V — значение N-ого числа, а $V1$ — значение $(N-1)$ числа.

Определим правила:

1. если N=0, то V=1, а $V1$ может быть любым;
2. если N=1, а Vl=0, то V=1;
3. иначе находятся $(N-1)$ и $(N-2)$ числа Фибоначчи в одном правиле и сумма этих чисел и будет N-ым числом Фибоначчи.

Код программы вычисления чисел Фиббоначи с использованием хвостовой рекурсии.

```
1   PREDICATES
2       fib2 (integer, integer, integer)
3   CLAUSES
4       fib2 (0, _, 0).
5       fib2 (1, 0, 1).
6       fib2 (N, V1, V) :-
7               N > 1, N1 = N - 1,
8               fib2 (N1, V2, V1),
```

```
9              V = V1 + V2.
10   GOAL
11        fib2 (4, X, Y) write (X),write (Y).
```

Оценка сложности вычислений. Максимальная глубина рекурсии: n. Общее число рекурсивных вызовов: n. Объем требуемой памяти на максимальной глубине рекурсии: $5{\times}(sizeof(int)){\times}n$.

2. Лабораторная работа 2. Управление списками.

Постановка задачи. Реализовать на языке ПРОЛОГ наборы правил для работы со списками:

– вывод списка на экран;
– добавление элемента в голову списка;
– добавление в хвост;
– добавление в определённую позицию;
– объединение 2 списков;
– удаление элемента из списка.

Представление списков на ПРОЛОГЕ основано на рекурсивном определении списка. Список либо пуст, либо состоит из головы и хвоста. Пустой список — [], не пустой список — $[X|Tail]$, где X — голова списка, а $Tail$ – оставшаяся часть.

Добавление элемента в начало списка.

Наиболее простой способ добавить элемент в список — это вставить его в самое начало так, чтобы он стал его новой головой. Если X — это новый элемент, а список, в который X добавляется — L, тогда результирующий список — это просто $[X|L]$. Графическое представление выполнения правила $[X|L]$ добавления элемента в голову списка приведено на рисунке 3.

Рисунок 3. Вызов правила добавления элемента в голову списка.

Таким образом, для того, чтобы добавить новый элемент в начало списка, не надо использовать никакой процедуры. Тем не менее, ес-

ли мы хотим определить такую процедуру в явном виде, то ее можно представить в форме такого факта: $ins(X, L, [X|L])$.

Добавление элемента в конец списка.

Поэлементно просматриваем исходный список. При этом каждый элемент из исходного списка поочерёдно добавляем в результирующий список. Как только в исходном списке достигнут конец — [], добавляем к результирующему списку элемент X.

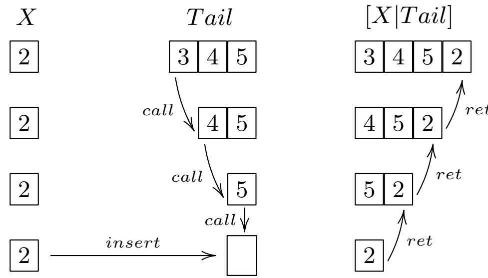

Рисунок 4. Рекурсивные вызовы правила добавления элемента в хвост списка.

Трассировка вызовов правила $ins(2, [3, 4, 5], list)$ добавления элемента в конец списка приведена на рисунке 4.

Добавление элемента в произвольную позицию списка.

Сначала выводим головной элемент списка, затем вызываем процедуру от хвоста. Таким образом, для вставки элемента в определённую позицию определим отношение $ins(L, X, Pos, L1)$. Здесь L — исходный список, $L1$ — результат, Pos — номер элемента, после которого добавляется новый элемент, X — новый элемент. Отношение ins выполняется двумя путями:

1. необходимая позиция не достигнута, тогда продвигаемся на следующую позицию;
2. позиция достигнута, тогда добавляем новый элемент.

Добавление элемента списка аналогично процедуре добавления в конец списка, однако, процесс рекурсивных вызовов прерывается при достижении позиции Pos, после чего в список добавляется новый элемент, а затем выполняется цепочка выходов из рекурсии.

Вывод списка на экран.

Сначала выводим головной элемент списка, затем вызываем процедуру от хвоста. Таким образом поэлементно перебирается весь список.

На рисунке 5 приведён пример трассировки правила вывода списка на экран $print_list([2, 3, 4, 5])$.

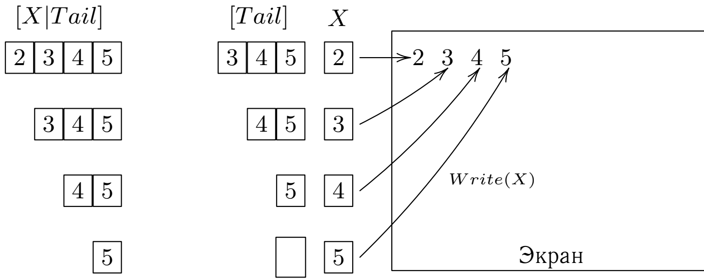

Рисунок 5. Рекурсивные вызовы правила вывода списка на экран.

Соединение двух списков.

Для сцепления списков определим отношение $plus(L1, L2, L3)$. Здесь $L1$ и $L2$ — два списка, а $L3$ — список, получаемый при их сцеплении. Например,

$$plus([a, b], [c, d], [a, b, c, d])$$

истинно, а

$$plus([a, b], [c, d], [a, b, a, c, d])$$

ложно.

Определение отношения $plus$, содержит два случая в зависимости от вида первого аргумента $L1$:

1. если первый аргумент пуст, тогда второй и третий аргументы представляют собой один и тот же список, назовём его L, что выражается следующим образом: $plus([\,], L, L)$,
2. если первый аргумент отношения $plus$ не пуст, то он имеет голову и хвост и выглядит так: $[X|L1]$.

Будем подставлять в переменную X значения из первого списка до тех пор, пока он весь не допишется в голову второго.

Пример трассировки правила $conc([2, 3, 4], [2, 5], X)$ соединения двух списков приведён на рисунке 6.

Удаление элемента из списка.

Удаление элемента X из списка L можно запрограммировать в виде отношения $del(X, L, L1)$,где $L1$ совпадает со списком L, у которого удалён элемент X. Отношение del можно определить следующим образом:

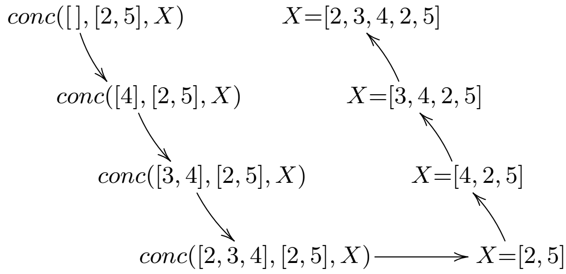

Рисунок 6. Рекурсивные вызовы правила соединения двух списков.

1. если X является головой списка, тогда результатом удаления будет хвост этого списка. Хвост так же просматривается на наличие элемента X;
2. если X находится в хвосте списка, тогда его нужно удалить оттуда. В результате из списка удаляются все элементы равные X.

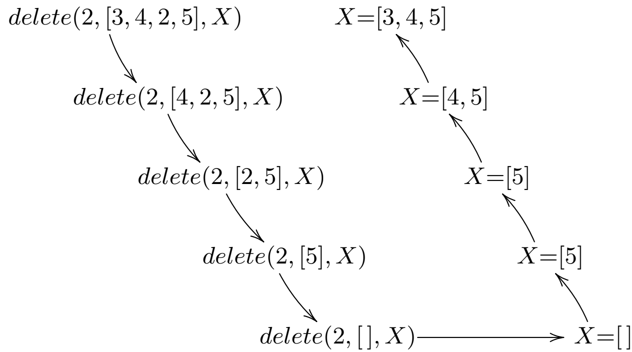

Рисунок 7. Рекурсивные вызовы правила удаления элемента из списка.

Пример трассировки правила $delete(2, [3, 4, 2, 5], X)$ удаления элемента и произвольной позиции списка приведён на рисунке 7.

Исходный текст программ.

Добавление элемента в начало списка.

```
1  DOMAINS
2      list = integer*
3      x = integer
4  PREDICATES
```

```
5       ins (x, list, list)
6   CLAUSES
7       ins (X, L, [X|L]).
```

Добавление элемента в конец списка

```
1   DOMAINS
2       list = integer*
3       x = integer
4   PREDICATES
5       ins (x, list, list)
6   CLAUSES
7       ins (X, [], [X]).
8       ins (X, [H|T1], [H|T2]) :- ins (X, T1, T2).
```

Добавление элемента в произвольную позицию списка

```
1   DOMAINS
2       list= integer*
3       x= integer
4   PREDICATES
5       ins (list, x, x, list)
6   CLAUSES
7       ins (Tail, X, 0, [X|Tail]).
8       ins([Head|Tail], X, Pos,
9                [Head|Tail2]) :- Pos1 =Pos - 1, !,
10                                  ins (Tail, X, Pos1, Tail2).
```

Вывод списка на экран

```
1   DOMAINS
2       number_list = integer*
3   PREDICATES
4       print_list (number_list)
5   CLAUSES
6       print_list ([]).
7       print_list ([Head|Tail]) :- write (Head), nl,
8                                   print_list (Tail).
```

Соединение двух списков

```
1   DOMAINS
2       list = integer*
3   PREDICATES
4       plus (list, list, list)
5   CLAUSES
6       plus ([], L, L).
7       plus ([N|L1], L2, [N|L3]) :- plus (L1, L2, L3).
```

Удаление элементе из списка

```
1  DOMAINS
2      list = integer*
3      x = integer
4  PREDICATES
5      delete (x, list, list)
6  CLAUSES
7      delete (X, [], []).
8      delete (X, [X|T], Z) :-
9      delete (X, T, Z), !.
10     delete (X, [Y|T], [Y|T1]) :-
11     delete (X, T, T1).
```

Оценка сложности.

Добавление элемента в начало списка.

Время выполнения программы $O(1)$, так как выполняется всего одна операция присоединения головы к списку. Глубина рекурсии — 1. Процедура запускается только один раз. Память требуется на хранение двух списков и одного добавляемого элемента, то есть порядка n, где n — количество элементов в списке.

Добавление элемента в конец списка. Время выполнения программы $O(n)$, где n — количество элементов в списке. Глубина рекурсии — $(n+1)$, при прохождении по списку на одну позицию вперед программа вызывает себя один раз. Таким образом, получаем, что бы добраться до элемента n процедуре необходимо вызвать себя n раз, и ещё 1 раз требуется на добавление элемента в список.

Добавление элемента в произвольную позицию списка.

Сложность процедуры добавления элемента в произвольную позицию будет составлять $O(k)$, где k — позиция нового элемента. Глубина рекурсии — k. Пока не достигнута позиция, вызываем процедуру от хвоста, как только позиция достигнута, добавляем элемент в начало оставшегося хвоста. Как видим, процедура вызовет себя k раз.

Вывод списка на экран. В этом случае глубина рекурсии растёт линейно относительно количества элементов в списке. Глубина рекурсии равна n, где n — количество элементов в списке.

Соединение двух списков. Здесь процедура вызовет саму себя k раз, где k — количество элементов в списке $L1$. Получаем, глубина рекурсии — k, сложность — $O(k)$.

Удаление элемента из списка. Время выполнения программы $O(n)$, покажем это: на просмотр всего списка требуется $O(n)$ операций n длин списка. Для удаления элемента из списка требуется конечное постоянное число операций, и в результате получаем, что количество рекурсивных вызовов будет n.

3. Лабораторная работа 3. Задача о расстановке 8 ферзей.

Постановка задачи. Реализовать на языке ПРОЛОГ решение задачи о 8 ферзях: необходимо расставить на шахматной доске 8 ферзей, так чтобы они не били друг друга.

Алгоритм. Идея алгоритма состоит в последовательном просмотре всех вертикалей и поиске на них первой годной позиции — такой, чтобы она не находилась под ударом ранее расставленных фигур. Для этого:

1. для каждой горизонтали выбираем координату по вертикали;
2. проверяем не бьёт ли ферзь расставленных на предыдущих шагах ферзей;
3. если бьёт, берём следующую координату;
4. если не бьёт, то устанавливаем на эту позицию ферзя и переходим к следующей горизонтали.

Код программы получения первого решения:

```
1   DOMAINS
2       point = integer*
3       list = point*
4   PREDICATES
5       ferz (integer, integer, list, point)
6       choose (integer, point, point)
7       may (point, list)
8       notfree (point, point)
9       abst (integer, integer)
10      ferzi (list)
11  CLAUSES
12      ferz (0, _, [], [8,7,6,5,4,3,2,1]) :- !.
13      ferz (8, M, [[8,X]|A], C) :- L = M + 1,
14                                   ferz (7, L, A, B),
15                                   choose (X, B, C),
16                                   may ([8, X], A), !.
17  ferz (N, M, [[N, X]|A], C) :- K = N - 1, L = M + 1,
18                                ferz (K, L, A, B),
19                                choose (X, B, C),
20                                may ([N, X],A).
21  choose (X, [X|A], A).
22  choose (X, [Y|A], [Y|B]) :- choose (X, A ,B).
23  may (_, []) :- !.
24  may (A, [B|C]) :- not (notfree (A, B)),
25                    may (A, C).
26  notfree ([A, _], [A, _]) :- !.
27  notfree ([_, A], [_, A]) :- !.
28  notfree ([A, B], [C, D]) :- K = A - C, abst (K, E),
29                              L = B - D, abst (L, F),
```

```
30                                E = F.
31  abst (X, Y) :- X < 0, !, Y = -1 * X.
32  abst (X, X).
33  ferzi (A) :- ferz (8, 0, A, []).
```

Результат работы – список: $List = [6, 3, 5, 7, 1, 4, 2, 8]$. Графическая интерпретация расстановки ферзей приведена на рисунке 8.

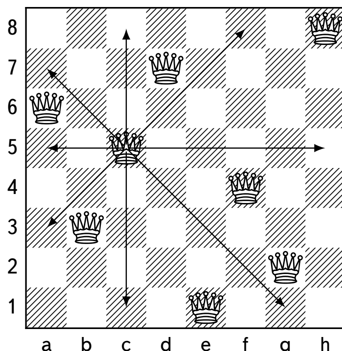

Рисунок 8. К решению задачи о расстановке 8 ферзей.

Для сокращения числа перебираемых вариантов можно полагать уже занятые диагонали. На i-м шаге мы рассматриваем $(8-i)$ диагоналей, среднее число клеток на диагонали — $\frac{i}{2}$, число вариантов сокращается на $(8-i)\times\frac{i}{2}$. Таким образом, получаем $O(n^2)$, что позволяет значительно ускорить поиск решения.

Код программы получения всех 92-х решений:

```
 1  DOMAINS
 2      point = integer*
 3      list = point*
 4  PREDICATES
 5      ferz (integer, integer, list, point)
 6      choose (integer, point, point)
 7      may (point, list)
 8      notfree (point, point)
 9      abst (integer, integer)
10      ferzi (list)
11  CLAUSES
12      ferz (0, _, [], [8,7,6,5,4,3,2,1]) :- !.
13      ferz (N, M, [[N, X]|A], C) :- K = N - 1,
14                                    L = M + 1,
15                                    ferz (K, L, A, B),
16                                    choose (X, B, C),
```

```
17                                   may ([N, X], A).
18  choose (X, [X|A], A).
19  choose (X, [Y|A], [Y|B]) :- choose (X, A ,B).
20  may (_, []) :- !.
21  may (A, [B|C]) :- not (notfree (A, B)), may (A, C).
22  notfree ([A, _], [A, _]) :- !.
23  notfree ([_, A], [_, A]) :- ! .
24  notfree ([A, B], [C, D]) :- K = A - C, abst (K, E),
25                              L = B - D, abst (L, F), E = F.
26  abst (X, Y) :- X < 0, !,
27                 Y = -1 * X.
28  abst (X, X).
29  ferzi (A) :- ferz(8, 0, A, []).
```

Оценка сложности алгоритма:

1. Максимальная глубина рекурсии: ferzi — 8, choose — 8, may — 8.
2. Общее число рекурсивных вызовов: функции ferzi и choose/may образуют 2 вложенных цикла, таким образом сложность алгоритма квадратичная.
3. Объём памяти:

ferzi — $4(4(1+2+3+\ldots+8)+3)=4(\frac{4\times8\times(8-1)}{2}+3) = 460$,

choose — $4(4(1+2+3+\ldots+8)+2)=4(\frac{2\times8\times(8-1)}{2}+2) = 232$,

may — $4(4(1+2+3+\ldots+8)+4)=4(\frac{2\times8\times(8-1)}{2}+4) = 128$.

На каждом шаге мы рассматривали поле размером $(8-k)\times(8-k)$, где k — число уже расставленных ферзей. Таким образом, на каждом шаге мы имеем $(8-k)^2$ способов расстановки.

Общее число вариантов:

$$\sum_{k=0}^{7}(8-k)^2 = \sum_{i=1}^{8} i^2.$$

Следует отметить, что существуют более эффективные методы сокращения числа операций, так как многие финальные позиции центрально или зеркально симметричны. Всего существует 12 групп уникальных решений, а значит все остальные решения могут быть получены путём зеркального отражения и поворотов одного из решений из каждой группы.

4. Лабораторная работа 4. Задача об обходе шахматного поля конем.

Постановка задачи. Необходимо реализовать на языке программирования ПРОЛОГ алгоритм решения задачи обхода шахматной доски конём. Имеется поле размера $N \times N$ ячеек. Необходимо обойти все поле шахматным конём так, что бы в каждой из ячеек побывать только один раз и побывать в каждой клетке.

Алгоритм. Производим обход поля следующим образом: делаем один из возможных ходов, затем проверяем, не включён ли этот ход уже в список ходов, если нет, то вносим его в список и ищем следующий, если внесён, то смотрим, какие ещё ходы можно сделать. Если из текущей позиции нельзя сделать ни одного хода, то отказываемся от этого хода и переходим к предыдущей позиции. В ходе работы алгоритма контролируем число посещённых клеток оценкой длины списка, когда длина списка станет равной 64, выводим весь список и завершаем работу алгоритма. Возможные позиции перемещения коня приведены на рисунке 9.

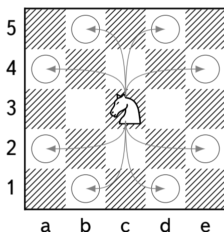

Рисунок 9. Возможные перемещения коня на следующем ходе.

Код программы:

```
1   DOMAINS
2       pos = pos (integer, integer)
3       list = pos*
4   PREDICATES
5       HorseStep (pos, pos)
6       moves (integer, integer)
7       OnBoard (integer)
8       HorsePath (list)
9       extmoves (integer, integer)
```

```
10 CLAUSES
11     % moves & extmoves определяют возможный ход конем
12     moves (2, 1).
13     moves (2, -1).
14     moves (-2, 1) .
15     moves (-2, -1) .
16     extmoves (Dx, Dy) :-
17     moves (Dx, Dy);
18     moves (Dy, Dx).
19     %проверка находится ли конь на доске
20     OnBoard (K) :- 0 < K, K < 9.
21     %шаг коня - вычисляем ходы из данной позиции
22     HorseStep (pos (X, Y), pos(X1, Y1)) :-
23                                          extmoves (Dx, Dy),
24                                          X1 = X + Dx,
25                                          OnBoard (X1),
26                                          Y1 = Y + Dy,
27                                          OnBoard (Y1).
28     %обход доски конем
29     HorsePath([pos (_,_)]).
30     HorsePath([pos (X,Y), pos(X1,Y1)|F]) :-
31                                    HorseStep (pos(X,Y),
32                                    pos(X1,Y1)),
33                               HorsePath([pos(X1,Y1)|F]).
```

Оценка сложности алгоритма: $8 + 8 * 7^{(n-l)}$, где n — количество шагов в пути. Максимальная глубина рекурсии: n^2, где n — размер поля. Количество вызовов:

$$\sum_{i=1}^{n^2}(1 - \frac{h}{n^2}) \times (2 \times 4 + 2 \times 24 + 8 \times 36).$$

5. Лабораторная работа 5. Сортировки списков.

Постановка задачи. Реализовать на языке программирования ПРОЛОГ следующие варианты сортировок списков:

– пузырьковая;
– вставками;
– быстрая.

Описание алгоритмов.

Пузырьковая сортировка

Найти в списке два смежных элемента X и Y, таких, что $X > Y$, и поменять их местами, получив тем самым новый список, затем отсортировать получившийся список. Если в списке нет ни одной пары

смежных элементов X и Y, таких, что $X > Y$, то считать, что список уже отсортирован.

Сортировка вставками

Для того чтобы упорядочить не пустой список $L = [X|\text{Хвост}]$, необходимо:

1. упорядочить хвост списка L;
2. вставить голову X списка L в упорядоченный хвост, поместив ее в такое место, чтобы получившийся список остался упорядоченным. Список отсортирован.

Быстрая сортировка

Для того, чтобы упорядочить не пустой список L, необходимо:

1. Удалить из списка L какой-нибудь элемент X и разбить оставшуюся часть на два списка, называемые Меньш и Больш, следующим образом: все элементы большие, чем X, принадлежат списку Больш, остальные - списку Меньш;
2. Отсортировать список Меньш, результат — список УпорМеньш;
3. Отсортировать список Больш, результат — список УпорБольш;
4. Получить результирующий упорядоченный список как конкатенацию списков УпорМеньш и $[X|\text{УпорБольш}]$.

Исходный текст программ.

```
1   bigger (X, Y) :- X > Y.
2   % Пузырьковая сортировка
3   puzirek (X, Y) :-
4                    change (X, List2), !,
5   puzirek (List2, Y.
6   puzirek (X, X).
7   change ([X, YID], [Y, XID]) :-
8                     bigger (X, Y).
9   change ([K|D], [K|D1]) :-
10                    change (D, D1).
11  % Сортировка вставками
12  vstavki ([], []).
13  vstavki ([X|Y], J) :-
14                     vstavki (Y, K),
15                     ins (X, K, J).
16  ins (X, [Y|J], [Y|J1]) :-
17                     bigger (X, Y), !,
18                     ins (X, J, J1).
19                     ins (X, J, [X|J]).
20  % Быстрая сортировка
21  bistraya ([], []).
22  bistraya ([X|Tail], L1) :-
23                      razbienie (X, Tail, M, B),
24                      bistraya (M, M1),
25                      bistraya (B, B1),
```

```
26                        plus (M1, [X|B1], L1).
27  razbienie (_, [], [], []).
28  razbienie (X, [Y|Tail],[Y|Mensh], Bolsh) :-
29                               bigger (X, Y), !,
30                    razbienie (X, Tail, Mensh, Bolsh).
31  razbienie (X, [Y|Tail], Mensh, [Y|Bolsh]) :-
32                    razbienie(X,Tail,Mensh,Bolsh).
```

Оценка сложности.

Пузырьковая сортировка

Алгоритм имеет квадратичную сложность $O(n^2)$, это достигается за счёт того, что каждый элемент сравнивается с каждым. Таким образом получаем, если у нас в списке п элементов, необходимо будет проделать n^2 операций. Глубина рекурсии растёт квадратично относительно количества элементов списка.

Сортировка вставками

Алгоритм так же имеет квадратичную сложность, на рисунке 10 изображена трассировка вызовов правил сортировки вставками: в рекурсивной ветви от исходного списка отделяется значение X, а при возврате просматривается с результатом и в него вставляется значение ранее отделённого элемента. На рекурсивном спуске мы отделяем голо-

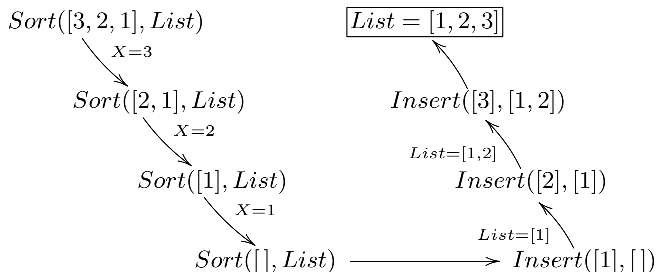

Рисунок 10. Рекурсивные вызовы программы сортировки вставками.

ву от списка до тех пора, пока не останется пустой список. На возврате мы вставляем голову списка в упорядоченный список, причём вставляем элемент так, чтобы упорядоченность не нарушалась. Таким образом получаем, сложность $O(n^2)$.

Быстрая сортировка

Эта сортировка является самой быстрой из всех трёх сортировок. Выполняется за самое меньшее время $O(n \times log(n))$, где n — количество элементов в списке. Откуда берётся такая цифра: $log(n)$ мы получаем, когда разбиваем список на части в процессе построения дерева, n

получаем когда распределяем элементы списка в меньшую и большую части. Однако, в случае если список уже упорядочен и упорядочен в обратном порядке, этот алгоритм теряет свою эффективность и работает за квадратичное время.

6. Лабораторная работа 6. Управление двоичными деревьями поиска.

Операции над двоичными деревьями поиска: создание, отображение на экране, вставка элемента на уровень листьев и на уровень узлов, в двоичное дерево поиска, поиск элемента в дереве.

Постановка задачи. Реализовать вычисление на языке программирования ПРОЛОГ операции с деревьями:
- отображение дерева на экране;
- вставка элементов на уровень листьев;
- вставка элементов на уровень узлов;
- поиск элемента в дереве.

Описание алгоритма. Двоичное дерево либо пусто, либо состоит из трёх частей:
- корень;
- левое поддерево;
- правое поддерево.

В корне хранится значение элемента дерева, а поддеревья должны сами быть двоичными деревьями. На рисунке 11 показано представление списка $[a, b, c, d]$ двоичным деревом. Элементы множества хранятся в виде вершин дерева. Пустые поддеревья на рисунке не показаны. Например, вершина b имеет два поддерева, причём оба пусты.

Отображение дерева на экране в наглядной форме происходит по следующим правилам:
1. отобразить правое поддерево с отступом на расстояние H, где H равно глубине дерева;
2. отобразить корень без отступа;
3. отобразить левое поддерево с отступом на расстояние H, где H равно глубине дерева.

Вставка элементов на уровень листьев.

Вставка элемента на уровень листа происходит по следующим правилам:
1. если дерево пусто, ставим X вместо корня;

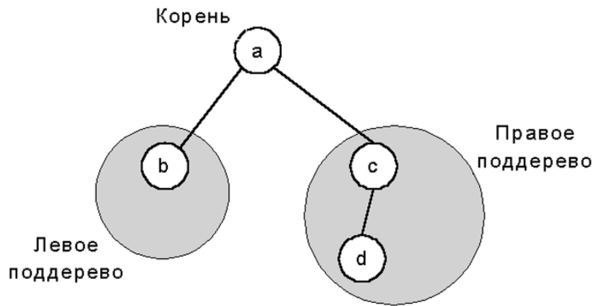

Рисунок 11. Представление списка $[a, b, c, d]$ в виде двоичного дерева.

2. если X равен корню дерева, то элемент уже есть и дополнительно добавлять его не требуется;
3. если X меньше корня дерева, то добавить X в левое поддерево;
4. если X больше дерева, то добавить X в правое поддерево.

Вставка элементов на уровень узлов.

Для того чтобы добавить X в двоичный справочник D, необходимо одно из двух:

1. добавить X на место корня дерева так, что X станет новым корнем или;
2. если корень больше, чем X, то внести X в левое поддерево, иначе в правое поддерево.

Трудным моментом здесь является введение X на место корня. Сформулируем эту операцию в виде отношения добкор(Д, Х, Х1) где X — новый элемент, вставляемый вместо корня в Д , а Д1 — новый справочник с корнем X. На рисунке 12 показано, как соотносятся Х, Д и Д1. Остаётся вопрос: что собой представляют поддеревья $L1$ и $L2$.

1. $L1$ и $L2$ — двоичные справочники;
2. множество всех вершин, содержащихся как в $L1$, так и в $L2$, совпадает с множеством вершин справочника L;
3. все вершины из $L1$ меньше, чем X; все вершины из $L2$ больше, чем X.

Поиск элемента в дереве.

Поиск элемента в дереве происходит по следующим правилам:

1. если X является корнем дерева, то возвратить $TRUE$;
2. если X меньше корня дерева, то искать X в левом поддереве;
3. если X больше корня дерева, то искать X в правом поддереве.

Удаление элемента из дерева.

Удаление элемента из дерева происходит по следующим правилам:

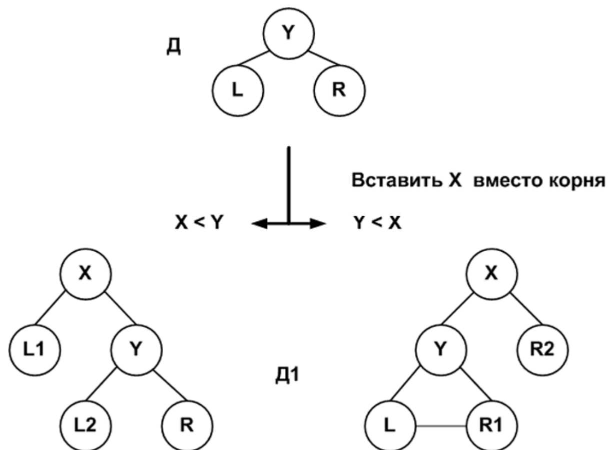

Рисунок 12. Внесение X в двоичный справочник в качестве корня.

1. находим удаляемый элемент;
2. ставим на его место левый элемент правого поддерева или правый элемент левого поддерева;
3. удаляем левый или правый элемент.

Исходный текст программы.

```
1   DOMAINS
2       i = integer
3       tree = nil;
4       tree (tree, integer, tree)
5   PREDICATES
6       greater (i, i)
7       intree (i, tree)
8       addlist (tree, i, tree)
9       ins (tree, i, tree)
10      insroot (tree, i, tree)
11      showtree (tree)
12      show (tree, i)
13      tab (i)
14      del (tree, i, tree)
15      delmin (tree, i, tree)
16  CLAUSES
17      %проверка наличия элемента в дереве
18      intree (X, tree (_, X, _)).
19      intree (X, tree (L, H, _)) :- greater (H, X),
20                                    intree (X, L).
21      intree (X, tree (_, H, R)) :- greater (X, H),
```

```
22                                     intree (X, R).
23        %добавление элемента на уровень листьев
24        addlist (nil, X, tree (nil, X, nil)).
25        addlist (tree (L, X, R), X, tree (L, X, R)).
26        addlist (tree (L, Root, R), X, tree (L1, Root, R)) :-
27                          greater (Root, X), addlist (L, X, L1).
28        addlist (tree (L, Root, R), X, tree (L, Root, R1)) :-
29                             greater (X, Root),
30                             addlist (R, X, R1).
31        %добавление элемента на уровень узлов
32        ins (T, X, T1) :- insroot (T, X, T1).
33        ins (tree (L, Y, R), X, tree (L1, Y, R)) :-
34                                                 greater (Y, X),
35                                                 ins (L, X, L1).
36        ins (tree (L, Y, R), X, tree (L, Y, R1)) :-
37                                                 greater (X, Y),
38                                                 ins (R, X, R1).
39        insroot (nil, X, tree (nil, X, nil)).
40        insroot (tree (L, Y, R), X,
41                           tree (L1, X, tree (L2, Y, R))) :-
42                               greater (Y, X),
43                               insroot (L, X, tree (L1, X, L2)).
44        insroot (tree (L, Y, R), X,
45                           tree (tree (L, Y, R1), X, R2)) :-
46                               greater (X, Y),
47                               insroot (R, X, tree (R1, X, R2)).
48        %вывод дерева на экран
49         showtree (T) :- how (T, 0).
50                         show (nil, _).
51         show (tree (L, X, R), I) :-
52                         I1 = I + 5,
53                         show (R, I1),
54                         tab (I),
55                         write (X), nl,
56                         show (L, I1).
57          tab (O).
58          tab (I) :- write (" "),
59                     I1 = I - 1,
60                     tab (I1).
61          %сравнение элементов
62          greater (A, B) :- A > B.
63          %удаление из дерева
64          del (tree (nil, X, R), X, R).
65          del (tree (L, X, nil), X, L).
66          del (tree (L, X, R), X, tree (L, Y, R1)) :-
67                                   delmin (R, Y, R1).
68          del (tree (L, Root, R), X, tree (L1, Root, R)) :-
69                                   greater (Root, X),
70                                   del (L, X, L1).
71          del (tree (L, Root, R), X, tree(L, Root, R1)) :-
72                                   greater (X, Root),
```

```
73                                del (R, X, R1).
74          delmin (tree (nil, Y, R), Y, R).
75          delmin (tree (L, Root, R), Y, tree (L1, Root, R)) :-
76                                delmin (L, Y, L1).
```

Оценка сложности

Отображение дерева на экране.

Необходимо обойти все элементы дерева и каждый отобразить на экране, получается линейная сложность. Для процедуры определения отступа, сложность определяется глубиной элемента дерева, какова глубина элемента, такова и глубина рекурсии в процедуре tab.

Вставка элементов на уровень листьев.

Если дерево сбалансировано, то эта операция потребует $log(N)$ рекурсивны вызовов. Если дерево не сбалансировано, то потребуется количество вызовов равное глубине ветви, в которую вставляется элемент. Это число может быть в диапазоне от 1, когда поддерево пустое до N, когда поддерево вырождено в список.

Вставка элементов на уровень узлов.

Для случая добавления в корень число вызовов процедуры пропорционально глубине дерева. Для сбалансированного дерева это число составит $log(N)$, так как процедура делит поддерево на две части, затем вызывается процедура от одного из поддеревьев, которое потом тоже делится на две части. Процедура повторяется до достижения максимальной глубины дерева. В итоге и получаем число вызовов пропорциональное $log(N)$.

Поиск элемента в дереве.

Глубина рекурсии данной процедуры определяется глубиной искомого элемента. Количество рекурсивных вызовов пропорционально глубине элемента.

Удаление элемента из дерева.

Сложность определяется двумя функциями: поиска искомого элемента и заменой корня. Обе эти функции имеют логарифмическую сложность. Сумма логарифмов — тоже логарифм, получаем итоговую сложность — логарифмическая.

7. Лабораторная работа 7. Операции на графе.

Нахождение всех ациклических путей на графе. Построение гамильтонова цикла графа. Нахождение пути минимальной стоимости.

Постановка задачи. Реализовать на языке ПРОЛОГ следующие функции для работы с графами:

– нахождение ациклических путей в графе;
– построение гамильтонова цикла;
– нахождение ациклического пути минимальной стоимости.

Представление графа в ПРОЛОГе.

Граф задаётся списком вершин и списком рёбер. В языке ПРОЛОГ граф можно задать следующим образом:

```
1       graph = graph (vertexlist, edgelist)
2       vertexlist = vertex*
3       vertex = integer
4       edgelist = edge*
5       edge = edge (vertex, vertex)
```

Реализация функций:

Нахождение ациклических путей в графе.

Для нахождения ациклического пути P между A и B в графе необходимо:

1. если A=B, то положить P=[A];
2. иначе найти ациклический путь $P1$ из произвольной вершины Y в B, а затем найти путь из A в Y, не содержащий вершин из $P1$.

Код программы:

```
1   DOMAINS
2       graph = graph (vertexlist, edgelist)
3       vertexlist = vertex*
4       vertex = integer
5       edgelist = edge*
6       edge = edge (vertex, vertex)
7   PREDICATES
8       ingraph (edge, edgelist)
9       ingraph1 (vertex, vertexlist)
10      conn (vertex, vertex, graph)
11      road (vertex, vertex, graph, vertexlist)
12      road1 (vertex, vertexlist, graph, vertexlist)
13  CLAUSES
14      % Поиск заданного ребра в списке ребер
15      ingraph (edge (X, Y), [edge (X, Y)|_]).
16      ingraph (edge (X, Y), [edge (_, _)|T]) :-
17      ingraph (edge (X, Y), T).
18      % Поиск вершины в списке вершин
19      mgraph1 (X,[X|_]).
20      ingraph1 (X, [_|L]) :- ingraph1 (X, L).
21      %Устанавливаем, являются ли вершины смежными или нет
22      conn (X, Y, graph (V, R)) :- ingraph (edge (X, Y), R);
23      ingraph (edge (Y, X), R).
24      % Ищем ациклический путь, начиная с конца
25      road (A, Z, G, P) :- road1 (A, [Z], G, P).
```

```
26        % Если достигли начальной вершины - выдать результат
27        road1(A,[A|P1],_,[A|P1]).
28        %Если находимся в промежуточной вершине,
29        % ищем смежные с ней  вершины, не
30        %принадлежащие пути чтобы избежать циклов
31        road1 (A, [Y|P1], G, P) :- conn (X, Y, G),
32                                   not (ingraph1 (X, P1)),
33                                   road1 (A, [X, Y|P1], G, P).
```

Оценка сложности алгоритма: максимальная глубина рекурсии — n. Сложность порядка - $O(n^2)$ оценка сверху, где n — число вершин в графе.

Построение гамильтонова цикла

Гамильтоновым циклом называется ациклический путь, проходящий через все вершины графа.

Код программы:

```
1   DOMAINS
2       graph = graph (vertexlist, edgelist)
3       vertexlist = vertex*
4       vertex = integer
5       edgelist = edge*
6       edge = edge (vertex, vertex)
7   PREDICATES
8       Hamilton (graph, vertexlist, vertex)
9       ingraph1 (vertex, vertexlist)
10      allnodes (vertexlist, vertexlist)
11  CLAUSES
12      % возвращает гамильтонов цикл, если он существует
13      Hamilton (graph (V, S), P, A) :-
14                          road (A, _, graph (_, S), P),
15                          allnodes (P, V).
16      % проверяет, что все вершины графа попали в путь
17      allnodes (_, []).
18      allnodes (P, [H|V]) :- ingraph1 (H, P),
19                             allnodes (P, V).
```

Оценка сложности алгоритма: максимальная глубина рекурсии — n. Сложность порядка - $O(n^3)$ (оценка сверху), где n — число вершин в графе.

Нахождение пути минимальной стоимости

Код программы:

```
1   DOMAINS
2       graph = graph (vertexlist, edgelist)
3       vertexlist = vertex*
4       vertex = integer
5       edgelist = edge*
6       edge = edge (vertex, vertex)
7   PREDICATES
```

```
8        ingraph (vertex, vertex, edgelist, integer)
9        ingraph1 (vertex, vertexlist)
10       conn (vertex, vertex, graph)
11       road (vertex, vertex, graph, vertexlist, integer)
12       road1 (vertex, vertexlist, integer,
13                                 graph, vertexlist, integer)
14   CLAUSES
15       % проверяет наличие ребра между X и Y
16       ingraph (X, Y, [edge (_, _, _)|T], Z) :-
17                                      ingraph (X, Y, T, Z).
18       % проверяет наличие вершины в пути
19       ingraph1 (X, [X|_]).
20       ingraph1 (X, [_|L]) :- ingraph1 (X, L).
21       % устанавливаем, являются ли вершины смежными, или нет
22       conn (X, Y, Z, graph (_, R)) :- ingraph (X, Y, R, Z);
23                                        ingraph (Y, X, R, Z).
24       % ищет ациклические пути между двумя вершинами
25       road (A, Z, G, P, C) :- road1 (A, [Z], 0, G, P, C).
26       road1 (A, [A[P1], C1, _, [A|P1],C1).
27       road1 (A, [Y|P1], C1, G, P, C) :-
28                              not (ingraph1 (X, P1)),
29                              C2 = C1 + Z,
30                              road1 (A, [X, Y|P1], C2, G, P, C).
31       ingraph (X, Y, [edge (X, Y, Z)|_], Z).
32       % выбирает путь с минимальной стоимостью
33       conn (X, Y, Z, G), sdf (A, B, G, MC1) :-
34                                         road (A, B, G, _, MC),
35                                         MC > MC1.
36       % ищет ациклический путь между двумя вершинами
37       % с минимальной стоимостью
38       minroad (A, B, G, MP, MC) :- road (A, B, G, MP, MC),
39                                    sdf (A, B, G, MC).
```

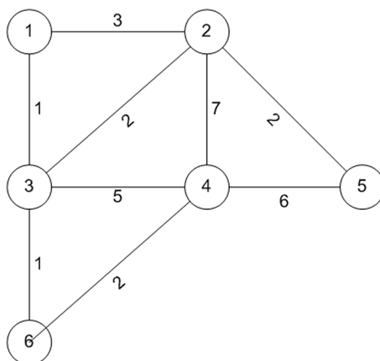

Рисунок 13. Пример взвешенного неориентированного графа.

Результаты поиска путей для графа, изображённого на рисунке 13:

все пути из 1 в 5 вершину:

```
1  1-2-4-5 V=16
2  1-3-2-4-5 V=16
3  1-3-4-5 V=12
4  1-2-3-4-5 V=16
5  1-3-6-4-5 V=10 - минимальный
6  1-2-3-6-4-5 V=14
```

все пути из 1 в 6 вершину:

```
1  1-3-6 V=2 - минимальный
2  1-2-3-6 V=6
3  1-2-4-3-6 V=16
4  1-2-4-6 V=12
5  1-3-2-4-6 V=12
6  1-3-4-6 V=8
7  1-2-3-4-6 V=12
```

Оценка сложности алгоритма: максимальная глубина рекурсии — n, сложность порядка — $O(n^3)$ (оценка сверху), где n — число вершин в графе.

8. Лабораторная работа 8. Построение остовных деревьев графа.

Постановка задачи. Нахождение остовного дерева минимальной стоимости. Реализовать на языке ПРОЛОГ следующие функции для работы с графами:

– нахождение всех остовных деревьев;

– нахождение остовного дерева минимальной стоимости.

Представление графа в ПРОЛОГе. Граф задаётся списком вершин и списком рёбер. В языке ПРОЛОГ граф можно задать следующим образом:

```
1      graph = graph (vertexlist, edgelist)
2      vertexlist = vertex*
3      vertex = integer
4      edgelist = edge*
5      edge = edge (vertex, vertex)
```

Граф называется связным, если между любыми двумя его вершинами существует путь. Пусть $G = (V, E)$ — связный граф с множеством вершин V и множеством ребер E. Остовное дерево графа G — это связный граф $T = (V, E')$, где E' — такое подмножество E, что T — связный граф и в нем нет циклов.

Реализация функций.

Нахождение всех остовных деревьев.

Алгоритм:

Начинаем с пустого множества рёбер. Постепенно добавляем рёбра, следя за тем, чтобы не образовывались циклы. Продолжаем этот процесс до тех пор, пока не обнаружится, что больше нельзя присоединить ни одного ребра, поскольку новое ребро порождает цикл.

Исключить появление циклов можно, используя следующее правило: к множеству рёбер присоединяются только такие ребра, у которых одна вершина уже принадлежит дереву, а другая в него ещё не включена. При этом для взвешенного графа можно добавить стоимости к дугам графа не изменяя алгоритм обхода графа. Графическое представление формирования остовного дерева приведены на рисунке 14.

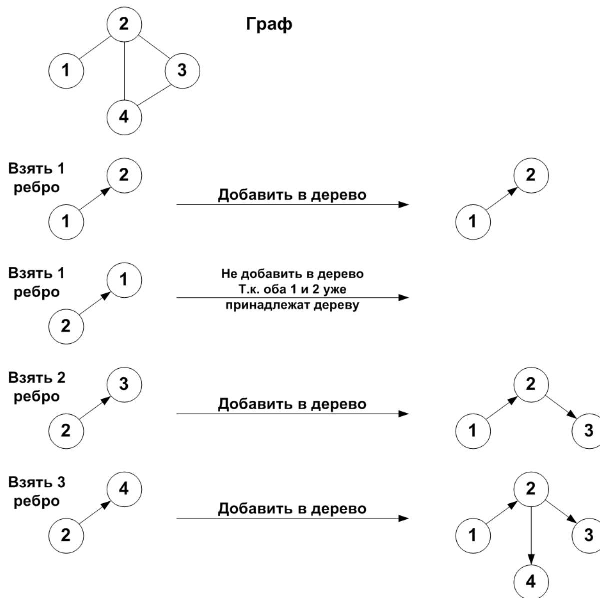

Рисунок 14. Графическое представление формирования остовного дерева графа.

1)Построение всех остовных деревьев из заданной вершины.

```
1  DOMAINS
2      i = integer
3      list = i*
4      rib = r (i, i)
5      rlist = rib*
```

```
6       graph = g (list, rlist)
7   PREDICATES
8       enlarge (rlist, rlist, graph)
9       spantree (graph, rlist)
10      insrib (rlist, rlist, graph)
11      adjoining (i, i, graph)
12      inrlist (rib, rlist)
13      rib_in_graph (rib, graph)
14      intree (i, rlist)
15      test (rlist)
16  CLAUSES
17      % остовное дерево Графа
18      spantree (G, T):- rib_in_graph (R, G),
19                        enlarge ([R], T, G).
20                        enlarge (T1, T, G) :-
21      % расширить дерево Tl до остовного дерева % T
22      insrib (T1, T2, G), enlarge (T2, T, G).
23      enlarge (T, T, G) :- not (insrib (T, _, G)).
24      % Добавление любого ребра приводит к циклу
25           % добавить ребро из графа и образовать новое дерево
26      insrib (T, [r (A, B)|T], G) :- adjoining (A, B, G),
27                                     intree (A, T),
28                                     not (intree (B, T)).
29      % проверить смежность между вершинами A и B
30      adjoining (A, B, G) :-      rib_in_graph (r (A, B), G);
31                              rib_in_graph (r (B, A), G).
32      % проверить, что принадлежит ли % нет A в дереву
33      intree (A, [r (A, _)|_]).
34      intree (A, [r (_, A)|_]).
35      intree (A, [_|L]) :- intree (A, L).
36      % взять ребро из графа
37      rib_in_graph (r (A, B), g (_, rlist)) :-
38                                inrlist (r (A, B), rlist).
39      inrlist (r (A, B), [r (A, B)|_]).
40      inrlist (r (A, B), [_|T]) :-
41      inrlist (r (A, B), T).
42      test (T) :-
43                 spantree (g ([1,2,3,4],
44                           [r(1,2),r(2,3),r(2,4),r(3,4)]),
45                           T).
```

Для нахождения дерева минимальной стоимости зададим новые правила, учитывающие вес построенного дерева:

```
1   DOMAINS
2   i = integer
3   list = i*
4   rib = r (i, i, i)
5   rlist = rib*
6   graph = g (list, rlist)
7   PREDICATES
```

```
8   enlarge (rlist, i, rlist, i, rlist)
9   spantree (rlist, rlist, i)
10  insrib (rlist, i, rlist, i, rlist)
11  adjoining (i, i, i, rlist)
12  inrlist (rib, rlist)
13  intree (i, rlist)
14  mense (rlist, i)
15  spantree_min (rlist, rlist, i)
16  test (rlist, i)
17  count_cost (rlist, i)
18  CLAUSES
19  count_cost ([], 0).                      % вычислить стоимость
20  count_cost ([r (A, B, CtAB)|T], Ct) :-
21  count_cost (T, Ct1),
22  Ct1 = Ct + CtAB.
23  spantree (G, T, Ct) :-          % Дер-остовное дерево Графа
24  inrlist (r (A, B, CtAB), G),
25  enlarge ([r (A, B, CtAB)], CtAB, T, Ct, G).
26  enlarge (T1, Ct1, T, Ct, G) :-
27  insrib (T1, Ct1, T2, Ct2, G),
28  enlarge (T2, Ct2, T, Ct, G).
29  enlarge (T, Ct, T, Ct, G) :-
30  not (insrib (T, Ct, _, _, G)).
31  % Добавление любого ребра приводит к циклу
32  insrib (T1, Ct1, [r (A, B, CtAB)|T1], Ct, G) :-
33  adjoining (A, B, CtAB, G),
34  Ct = Ct1 + CtAB,
35  intree (A, T1),
36  not (intree (B, T1)).
37  adjoining (A, B, CtAB, G) :-
38  inrlist (r (A, B, CtAB), G);
39  inrlist (r (B, A, CtAB), G).
40  intree (A, T) :-
41  adjoining (A, _, _, T).
42  inrlist (Rib, [Rib|_]).
43  inrlist (Rib, [_|T]) :-
44  inrlist (Rib, T).
45  mense (G, Ctmin) :-
46  spantree (G, _, Ct),
47  Ct < Ctmin.
48  spantree_min (G, Tmin, Ctmin) :-
49  spantree (G, Tmin, Ctmin),
50  not (mense (G, Ctmin)).
51  test (Tmin, Ctmin) :-
52  spantree_min([r(1,2,1),r(2,3,2),r(2,4,1),r(3,4,1)],
53  Tmin, Ctmin).
```

9. Решение математической головоломки.

Постановка задачи. Необходимо реализовать на языке программирования ПРОЛОГ алгоритм решения математической головоломки. Известным примером числового ребуса является

```
  D O N A L D
+ G E R A L D
= R O B E R T
```

Задача состоит в том, чтобы заменить все буквы на цифры таким образом, чтобы вышеприведённая сумма была правильной. Разным буквам должны соответствовать разные цифры, иначе возможно тривиальное решение, например, все буквы можно заменить на нули.

Словесное описание алгоритма. Определим целевое отношение сумма($N1, N2, N$), где $N1, N2$ и N представляют три числа данного ребуса. Цель сумма($N1, N2, N$) достигается, если существует такая замена букв цифрами, что $N1 + N2 = N$.

Первым шагом к решению будет выбор представления чисел $N1, N2$ и N в программе. Один из способов — представить каждое число в виде списка его цифр. Например, число 255 будет тогда представляться списком [2,2,5]. Поскольку значения цифр нам не известны заранее, каждая цифра будет обозначаться соответствующей не инициализированной переменной. Используя это представление, мы можем сформулировать задачу так:

```
  [D, O, N, A, L, D]
+ [G, E, R, A, L, D]
= [R ,O, B, E, R, T]
```

Теперь задача состоит в том, чтобы найти такую конкретизацию переменных D, O, N для которой сумма верна. После того, как отношение сумма будет запрограммировано, задание для системы ПРОЛОГ на решение ребуса будет иметь вид

$$? - \textit{сумма}([D,\ O,\ N,\ A,\ L,\ D],\ [G,\ E,\ R,\ A,\ L,\ D],\ [R,\ O,\ B,\ E,\ R,\ T]).$$

Тогда каждое число в сумме может быть представлено в виде:

$$\textit{Число1} = [D11, D12, \ldots, D1i, \ldots]$$

$$\textit{Число2} = [D21, D22, \ldots, D2i, \ldots]$$

$$\textit{Число3} = [D31, D32, \ldots, D3i, \ldots]$$

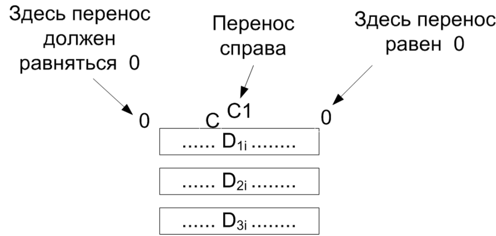

Рисунок 15. Графическая интерпретация переноса разряда в решении головоломкип.

Поразрядное сложение. Отношения в показанном i-м разряде такие:

$$D3i = (C1 + D1i + D2i)\, mod\, 10$$

$$C = (C1 + D1i + D2i)\, div\, 10$$

где, div — целочисленное деление, mod — остаток от деления.

Для определения отношения *сумма* над списками цифр нам нужно запрограммировать реальные правила суммирования в десятичной системе счисления. Суммирование производится цифра за цифрой, начиная с младших цифр в сторону старших, всякий раз учитывая цифру переноса справа. Необходимо также сохранять множество допустимых цифр, которые ещё не были использованы для конкретизации уже встретившихся переменных. Поэтому, вообще говоря, кроме трёх чисел $N1, N2$ и N в рассмотрении должна участвовать некоторая дополнительная информация, как показано на 15:

– перенос перед сложением;
– перенос после сложения;
– множество цифр, доступных перед сложением;
– оставшиеся цифры, не использованные при сложении.

Для формулировки отношения сумма мы снова воспользуемся принципом обобщения задачи: введём вспомогательное, более общее отношение *сумма1*. Это отношение будет иметь несколько дополнительных аргументов, соответствующих той дополнительной информации, о которой говорилось выше:

```
сумма1 ( N1, N2, N, C1, C, Цифры1, Цифры)
```

Здесь $N1, N2$ и N — три числа, как и в отношении *сумма*, $C1$ — перенос справа (до сложения $N1$ и $N2$), а C — перенос влево после сложения.

Если $N1$ и N удовлетворяют отношению *сумма*, то, как показано на рисунке 15, $C1$ и C должны быть равны 0. *Цифры1* — список цифр,

которые не были использованы для конкретизации переменных. Поскольку мы допускаем использование в отношении *сумма* любых цифр, ее определение в терминах отношения *сумма1* выглядит так:

```
сумма( N1, N2, N) :-
сумма1( N1, N2, N, 0, 0,
        [0,1,2,3,4,5,6,7,8,9], _).
```

Бремя решения задачи переложено теперь на отношение *сумма1*. Но это отношение является уже достаточно общим, чтобы можно было определить его рекурсивно. Без ограничения общности мы предположим, что все три списка, представляющие три числа, имеют одинаковую длину. Наш пример, конечно, удовлетворяет этому условию, но если это не так, то всегда можно приписать слева нужное количество нулей к более «короткому» числу.

Определение отношения *сумма1* можно разбить на два случая:

1. все три числа представляются пустыми списками. Тогда отношение *сумма1*$([\,],[\,],[\,],0,0,$*Циф*,*Циф*$)$.
2. Все три числа имеют какую-то самую левую цифру и справа от нее - остальные цифры. То есть, они имеют вид: $[D1|N1],[D2|N2],[D|N]$. В этом случае должны выполняться два условия:
   - 2.1. оставшиеся цифры, рассматриваемые как три числа $N1, N2$ и N, сами должны удовлетворять отношению *сумма1*, выдавая влево некоторый перенос $C2$ и оставляя некоторое подмножество неиспользованных цифр *Циф2*.
   - 2.2. Крайние левые цифры $D1, D2$ и D, а также перенос $C2$ должны удовлетворять отношению, показанному на рисунке 15: $C2, D1$ и $D2$ складываются, давая в результате D и перенос влево. Это условие в нашей программе формулируется в виде отношения *СуммаЦиф*.

Осталось только описать на ПРОЛОГЕ отношение *СуммаЦиф*. В его определении есть одна тонкая деталь, касающаяся применения металогического предиката *nonvar*. $D1, D2$ и D должны быть десятичными цифрами. Если хоть одна из этих переменных ещё не конкретизирована, её нужно конкретизировать какой-нибудь цифрой из списка *Циф2*. Как только такая конкретизация произошла, эту цифру нужно удалить из множества доступных цифр. Если $Dl, D2$ и D уже конкретизированы, тогда, конечно, ни одна из доступных цифр «потрачена» не будет. В программе эти действия реализуются при помощи недетерминированного вычёркивания элемента списка. Если этот элемент - не переменная, ничего не вычёркивается, так как конкретизации не было.

Исходный текст программы

```
1   DOMAINS
2       i = integer
3       list = integer*
4   PREDICATES
5       add (list, list, list)
6       add1 (list, list, list, i, i, list, list)
7       addnums (i, i, i, i, i, list, list)
8       delete (i, list, list)
9   CLAUSES
10      add (N1, N2, N) :-
11           add1 (N1, N2, N, 0, 0, [0,1,2,3,4,5,6,7,8,9], _).
12           add1 ([], [], [], 0, 0, Num, Num).
13      add1 ([D1|N1], [D2|N2], [D|N], C1, C, Num1, Num) :-
14           add1 (N1, N2, N, C1, C2, Num1, Num2),
15      addnums (D1, D2, C2, D, C, Num2, Num).
16      addnums (D1, D2, C1, D, C, Num1, Num):-
17                    delete (D1, Num1, Num2),
18                    delete (D2, Num2, Num3),
19                    delete (D, Num3, Num),
20                    S = D1 + D2 + C1,
21                    D = S mod 10,
22                    C = S div 10.
23      delete (A, L, L) :-
24             not (free (A)),!.
25      delete (A, [A|L], L).
26      delete (A, [B|L], [B|L1]) :-
27             delete (A, L, L1).
```

Пример:

```
GOAL:  DONALD + GERALD = ROBERT
```

В результате работы программы получим следующий результат:

```
D=5, O=2, N=6, A=4, L=8, G=1, E=9, R=7, B=3, T=0.
```

Вычислительная сложность алгоритма составляет $On \times k^2$, где n — число разрядов в складываемых числах, а k — основание системы исчисления.

10. Лабораторная работа 9. Задача о волке козе и капусте.

Постановка задачи. На одном берегу реки находятся волк, коза и капуста. Через реку на лодке курсирует перевозчик. В лодку может поместиться кроме перевозчика ещё одно животное или растение. Необходимо перевезти всех на другой берег в целости и сохранности, но

нельзя забывать о том, что волк ест козу, а коза капусту, конечно если на берегу нет перевозчика.

Идея реализации алгоритма состоит в переборе возможных состояний и продолжении вычислений только из тех из них, которые не завершают перевозку неудачей.

Алгоритм. Нахождение и вывод решения.

```
1   DOMAINS
2   % pvkk - структура-состояние, показывающая на каком берегу
3   % находятся перевозчик, волк, коза и капуста, соответственно
4      pvkk = pvkk (integer, integer, integer, integer)
5      list = pvkk*
6   PREDICATES
7       hod (pvkk, pvkk)
8       bad (pvkk)
9       inlist (pvkk, list)
10      mhod (list, list)
11  CLAUSES
12  % возможные ходы -перевозчик может переплыть пустым на др берег
13  % или взять еще кого-нибудь
14      hod (pvkk (1, A, B, C), pvkk (2, A, B, C)).
15      hod (pvkk (1, 1, B, C), pvkk (2, 2, B, C)).
16      hod (pvkk (1, A, 1, C), pvkk (2, A, 2, C)).
17      hod (pvkk (1, A, B, 1), pvkk (2, A, B, 2)).
18      hod (pvkk (2, A, B, C), pvkk (1, A, B, C)).
19      hod (pvkk (2, 2, B, C), pvkk (1, 1, B, C)).
20      hod (pvkk (2, A, 2, C), pvkk (1, A, 1, C)).
21      hod (pvkk (2, A, B, 2), pvkk (1, A, B, 1)).
22  % нельзя оставлять волка с козой и козу с капустой без присмотра
23      bad (pvkk (A, B, B, _)) :- A<>B.
24      bad (pvkk (A, _, B, B)) :- A<>B.
25  % проверка есть ли состояние уже в списке (для того, чтобы
26  % программа не зацикливалась)
27      inlist (pvkk (A, B, C, D), [pvkk (A, B, C, D)|_]).
28      inlist (A, [_|L]) :-
29            inlist(A,L).
30  % находим решение - последовательность перевозки, если состояние
31  % конечное, то заканчиваем поиск решения
32      mhod ([pvkk (2, 2, 2, 2)|L], [pvkk (2, 2, 2, 2)|L]).
33  % находим очередной ход, проверяем не ест ли кто-либо друг друга,
34  % проверяем не было это состояние уже раньше, и делаем ход дальше
35  % рекурсивно
36      mhod ([E1|L], L2) :-
37              hod (E1, N),
38              not (bad (N)),
39              not (inlist (N, L)),
40      mhod([N, E1|L], L2).
```

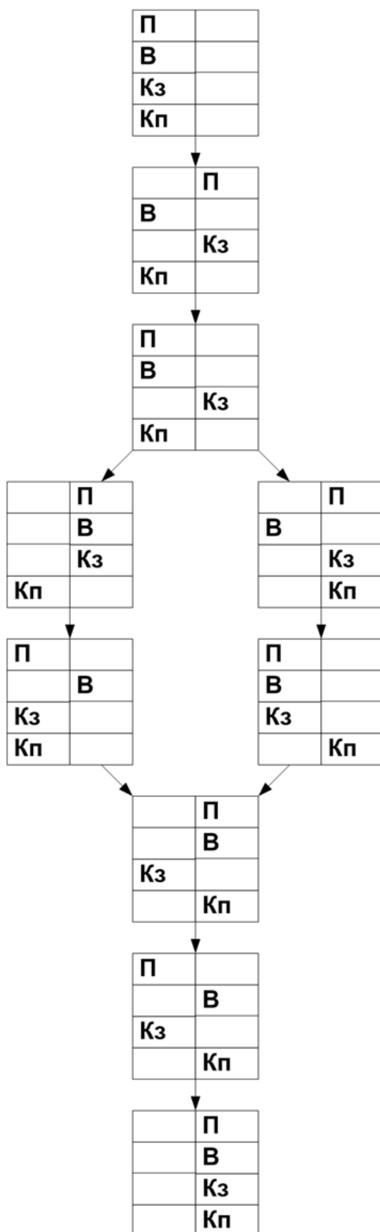

Рисунок 16. Графическое представление дерева решений графа в задаче о перевозчике. Тупиковые ветви не указаны.

Графическое представление фрагмента пространства состояний с переходами между решениями изображено на рисунке 16.

Алгоритм. Вывод всех промежуточных состояний.

```
1   DOMAINS
2       pvkk = pvkk (integer, integer, integer, integer)
3       list = pvkk*
4       PREDICATES
5       hod (integer, pvkk, pvkk)
6       bad (pvkk)
7       makehod (integer, list, list)
8       comp (pvkk, pvkk)
9       inlist (pvkk, list)
10      states (integer, list, list)
11      sol (list)
12  CLAUSES
13      hod (1, pvkk (1, A, B, C), pvkk (2, A, B, C)).
14      hod (2, pvkk (1, 1, B, C), pvkk (2, 2, B, C)).
15      hod (3, pvkk (1, A, 1, C), pvkk (2, A, 2, C)).
16      hod (4, pvkk (1, A, B, 1), pvkk (2, A, B, 2)).
17      hod (5, pvkk (2, A, B, C), pvkk (1, A, B, C)).
18      hod (6, pvkk (2, 2, B, C), pvkk (1, 1, B, C)).
```

Фрагмент пространства состояний задачи о волке, козе и капусте. Запрещённые состояния не показаны.

```
1   hod (7, pvkk (2, A, 2, C), pvkk (1, A, 1, C)).
2   hod (8, pvkk (2, A, B, 2), pvkk (1, A, B, 1)).
3   bad (pvkk (A, B, B, _)) :- A<>B.
4   bad (pvkk (A, _, B, B)) :- A<>B.
5   comp (pvkk (A, B, C, D), pvkk (A, B, C, D)).
6   inlist (pvkk (A, B, C, D), [pvkk (A, B, C, D)|_]).
7   inlist (A, [_|L]) :-
8   inlist (A, L).
9   states (8, [H|T], L2) :-
10  makehod (8, [H|T], L2).
11  states (I, [H|T], L2) :-
12  I < 8,
13  makehod (I, [H|T], L1),
14  J = I + 1,
15  states (J, L1, L2).
16  makehod (I, [H|T], L2) :-
17  hod (I, H, N),
18  not (inlist (N, T)),
19  not (bad (N)),
20  states (1, [N, H|T], L2), !;
21  L2 = [H|T].
22  sol (L) :-
23  states (1, [pvkk (1, 1, 1, 1)], L).
```

Оценка сложности. Вычислительная сложность алгоритма составляет $O2^n$, где n — число объектов для перемещения. Процедура поиска с возвратом позволяет сократить число перебираемых вариантов за счёт отсечения лишних ветвей.

11. Лабораторная работа 10. Задача о ханойской башне.

Постановка задачи. Необходимо реализовать на языке программирования ПРОЛОГ алгоритм решения задачи о ханойской башне.

Имеется три колышка 1, 2 и 3 и три диска a, b и c, причём, a — наименьший из них, а c — наибольший. Первоначально все диски находятся на колышке 1. Задача состоит в том, чтобы переложить все диски на колышек 3, На каждом шагу можно перекладывать только один диск, причём никогда нельзя помещать больший диск на меньший. Графическое представление исходного и финального состояний приведено на рисунке 17.

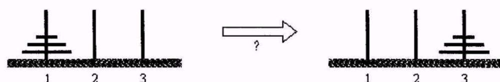

Рисунок 17. Представление исходного и финального состояний в задаче о ханойской башне.

Словесное описание алгоритма. Задачу можно рассматривать как задачу достижения следующих трех целей:

1. диск a — на колышек 3,
2. диск b — на колышек 3,
3. диск c — на колышек 3.

Проблема в том, что эти цели зависят друг от друга. Например, можно сразу переложить диск а на колышек 3, и первая цель будет достигнута. Но тогда две другие цели станут недостижимы, если только мы не отменим первое наше действие. К счастью, существует такой удобный порядок достижения этих целей, из которого можно легко вывести искомое решение.

Порядок этот можно установить при помощи следующего рассуждения: самая трудная цель —- это цель 3: Диск c — на колышек 3, потому что на диск c наложено больше всего ограничений. В подобных ситуациях часто срабатывает хорошая идея: пытаться достичь первой самую

трудную цель. Этот принцип основан на следующей логике: поскольку другие цели достигнуть легче – на них меньше ограничений, можно надеяться на то, что их достижение возможно без отмены действий на достижение самой трудной цели.

Применительно к нашей задаче это означает, что необходимо придерживаться следующей стратегии:

1. первой достигнуть цель «диск c — на колышек 3», а затем — все остальные цели. Но первая цель не может быть достигнута сразу, так как в начальной ситуации диск c двигать нельзя. Следовательно, сначала мы должны подготовить этот ход, и наша стратегия принимает такой вид:
   1.1. обеспечить возможность перемещения диска c с 1 на 3;
   1.2. переложить c с 1 на 3;
   1.3. достигнуть остальные цели — a на 3 и b на 3;
2. переложить c с 1 на 3 возможно только в том случае, если диск a и b оба надеты на колышек 2. Таким образом наша исходная задача перемещения a, b и c с 1 на 3 сводится к следующим трём подзадачам: для того, чтобы переложить a, b и c с 1 на 3, необходимо:
   2.1. переложить a и b с 1 на 2, и
   2.2. переложить c с 1 на 3, и
   2.3. переложить a и b с 2 на 3.

Задача 2 тривиальна и она решается за один шаг. Остальные две подзадачи можно решать независимо от задачи 2, так как диски a и b можно двигать не обращая внимание на положение диска c. Для решения задач 1 и 3 можно применить тот же самый принцип разбиения, тогда на этот раз диск b будет самым «трудным». В соответствии с этим принципом задача 1 сводится к трём тривиальным подзадачам: Для того, чтобы переложить a и b с 1 на 2, необходимо:

1. переложить a с 1 на 3, и
2. переложить b с 1 на 2, и
3. переложить a с 3 на 2.

Исходный текст программы.

```
1   DOMAINS
2       i = integer
3   PREDICATES
4       hanoy (i)
5       move (i, i, i, i)
6       output (i, i)
7   CLAUSES
8       hanoy (N) :- move (N, 1, 2, 3).
9       move (0, _, _, _) :- !.
10      move (N, A, B, C) :- X = N - 1,
11                           move (X, A, C, B),
```

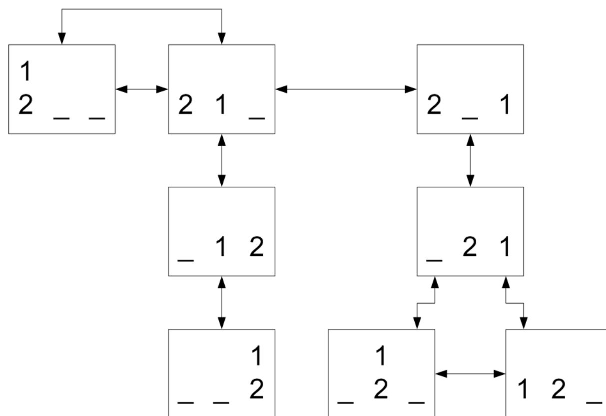

Рисунок 18. Фрагмент пространства состояний задачи о ханойской башне. Запрещённые состояния не показаны.

```
12        output (A, B), move (X, C, B, A).
13        output (X, Y) :- write (X, " => ", Y), nl.
```

Оценка сложности. Алгоритм имеет факториальную сложность. На первом шаге мы решаем задачу сложности N. На следующем шаге сложность составит $N-1$. На шаге k сложность будет $N-k$. Наконец, на шаге $N-1$ сложность будет 1. Что бы получить общую сложность алгоритма, перемножаем все эти числа и получаем в итоге $N!$.

На каждом шаге итерации хранятся четыре переменные: количество дисков на колышке, номера колышков. Таким образом, рост объёма использованной памяти составит $O(2^l)$, где l — глубина рекурсии отношения *move*.

Визуализация пространства состояний. Фрагмент пространства состояний задачи о ханойской башне для двух дисков показан на рисунке 18, запрещённые состояния не показаны. Полное пространство состояний приведено на рисунке 19.

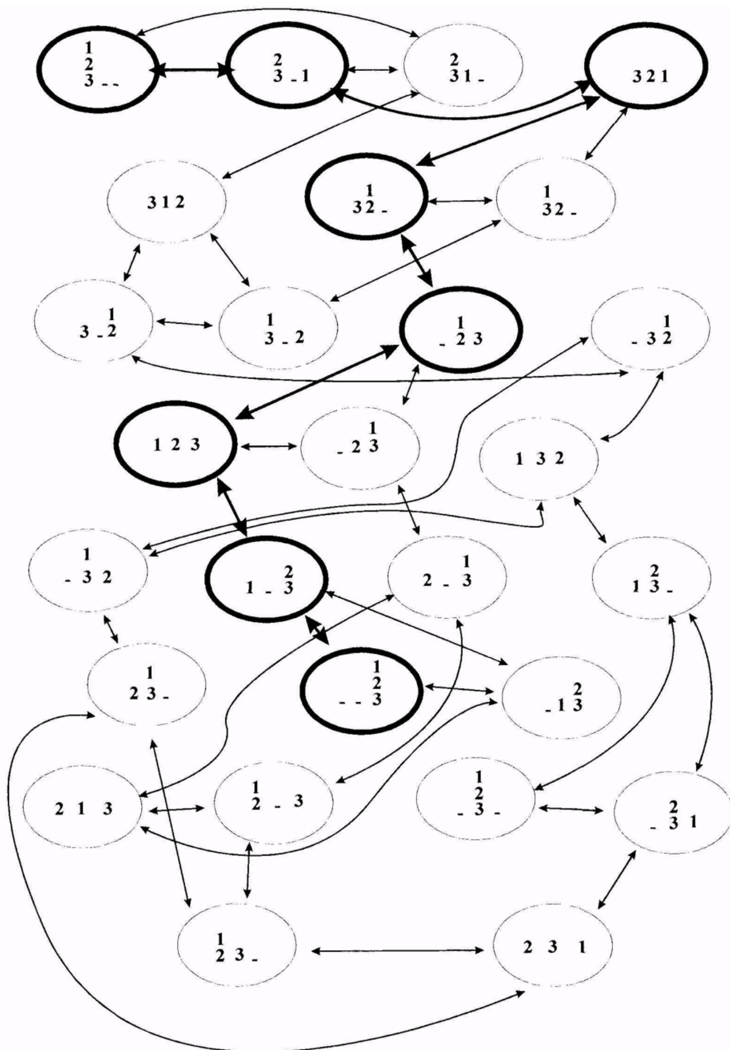

Рисунок 19. Пространство состояний задачи о ханойской башне.

12. Лабораторная работа 11. Задача оптимального распределения ресурсов вычислительной системы реального времени.

Постановка задачи. Имеется список задач, для каждой известна длительность выполнения и список задач, которые необходимо выполнить прежде, чем можем приступить к этой; для выполнения задач выделено некоторое количество процессоров. Необходимо определить порядок назначения задач каждому процессору так, чтобы общее время исполнения всех задач было минимальным.

Каждая задача может быть решена на любом процессоре, но в каждый данный момент каждый процессор решает только одну из задач. Между задачами существует отношение предшествования, определяющее, какие задачи должны быть завершены, прежде чем данная задача может быть запущена. Необходимо распределить задачи между процессорами без нарушения отношения предшествования, причём таким образом, чтобы вся совокупность задач была решена за минимальное время. Время, когда последняя задача в соответствии с выработанным планом завершает своё решение, называется временем окончания плана. Мы хотим минимизировать время окончания по всем возможным планам.

Описание алгоритма. Начинаем с пустого плана, с незаполненными временными промежутками для каждого процессора, и постепенно включаем в него задачи, одну за другой, пока все задачи не будут исчерпаны. Как правило, на каждом шагу мы будем иметь несколько различных возможностей, поскольку окажется, что одновременно несколько задач-кандидатов ждут своего выполнения. Таким образом, для составления плана потребуется перебор.

Теперь нам необходимо принять решение относительно представления проблемных ситуаций, частичных планов. Нам понадобится следующая информация:

1. список ждущих задач вместе с их временами выполнения;
2. текущая загрузка процессоров задачами;
3. время окончания частичного плана - технологическая компонента добавлена для удобства программирования.

Нашей эвристической функцией будет оптимистическая оценка времени окончания частичного плана с учётом всех ждущих задач. Оптимистическая оценка будет вычисляться в предположении, что два из ограничений, налагаемых на действительно корректный план, ослаблены:

1. не учитываются отношения предшествования;
2. делается не реальное допущение, что возможно распределённое выполнение задачи одновременно на нескольких процессорах, причём сумма времён выполнения задачи на процессорах равна исходному времени выполнения этой задачи на одном процессоре.

Код программы

```
1  % список задач, которые нужно решить перед заданной
2  preq = integer*
3  preqs = preq*
4  % задача: номер, длительность,
5  task = record (integer, integer, preq)
6  % требования
7  tasks = task*
8  % процессор: время освобождения и список
9  % задач, которые ему назначались, в
10 % обратном порядке; голова списка
11 % текущая задача
12 proc = record (integer, preq)
13 procs = proc*
14 mpc = record (procs, procs)
15 % МПЭВМ - списки работающих и
16 % простаивающих процессоров
17 result = record (integer, procs, procs)
18 % результат -  длительность и списки процессоров со
19 % списками назначенных задач
20 Реализация
21 % Выборка задач, готовых к выполнению
22 deboss (tasks, tasks)
23 % Задачи, готовые к выполнению будут иметь пустой
24 % список требований
25 deboss ([], []) :- !.
26 deboss ([record (I, T, [])|XS], [record (I, T, [])|QS]) :-
27 deboss (XS, QS), !.
28 deboss ([X|XS], QS) :- deboss (XS, QS).
29 % Удаление задачи из всех списков требований
30 removv (preq, integer, preq)
31 remov (tasks, integer, tasks)
32 % Удаление из одного списка требований
33 removv ([], _, []) :- !.
34 removv ([X|XS], X, S) :-
35 removv (XS, X, S), !.
36 removv ([Y|XS], X, [Y|S]) :-
37 removv (XS, X, S).
38 % Перебор списка неназначенных задач - вызов removv
39 % для каждой
40 remov ([], _, []) :- !.
41 remov ([record (I, T, X)|XS], N, [record (I, T, XX)|Q]) :-
42 remov (XS, N, Q),
```

```
43  removv (X, N, XX).
44  % Удаление задачи из всех списков неназначенных
45  takeout (tasks, integer, tasks)
46  takeout ([], _, []) :- !.
47  takeout ([record (I, _, _)|XS], I, S) :-
48  takeout (XS, I, S), !.
49  takeout ([X|XS], I, [X|S]) :-
50  takeout (XS, I, S).
51  % Добавление процессора в список простаивающих
52  setStop2 (procs, integer, procs)
53  setStop (procs, proc, procs)
54  % Устанавливаем всем простаивающим процессорам текущее
55  % время в  качестве времени освобождения - при назначении
56  % задачи мы прибавим к нему длительность задачи и
57  % получим реальное время освобождения
58  setStop2 ([], _, []) :- !.
59  setStop2 ([record (_, Q)|XQ], I, [record (I, Q)|XS]) :-
60  setStop2 (XQ, I, XS).
61  % Добавляем процессор в список и устанавливаем его
62  % время освобождения % всем остальным процессорам
63  setStop ([], Q, [Q]) :- !.
64  setStop (Q, record (I, L), [record (I, L)|XS]) :-
65  setStop2 (Q, I, XS).
66  % Добавление процессора в список работающих
67  desper (proc, procs, procs)
68  % Добавление происходит с сохранением упорядоченности
69  % по возрастанию времени освобождения
70  desper (Q, [], [Q]) :- !.
71  desper (record (I1, Q1), [record (I2, Q2)|QS],
72  [record (I1, Q1)|[record (I2, Q2)|QS]]) :-
73  I2>I1, !.
74  desper (Q, [W|QS], [W|WS]) :-
75  desper (Q, QS, WS).
76  % Подсчет времени выполнения последовательности
77  tempo (procs, integer)
78  % Поскольку все работающие процессоры упорядочены
79  % по возрастанию времени освобождения, время освобождения
80  % последнего и будет временем окончания
81  tempo ([record (I, _)], I) :- !.
82  tempo ([X|XS], I) :-
83  tempo (XS, I).
84  % Выбор результата с меньшим временем выполнения
85  pickless (result, result, result)
86  pickless (record (-1, _, _), Q, Q) :- !.
87  pickless (Q, record (-1, _, _), Q) :- !.
88  % Если одна из них - результат тупиковой ветви, то
89  % выбираем другую
90  pickless (record (I1, W1, S1), record (I2, W2, S2),
91  record (I3, W3, S3)) :-
92  I1 < I2, !.
93  pickless (_, Q, Q).
```

```
94   % Основной цикл
95   pktask (tasks, tasks, mpc, result)
96   wrkpush (tasks, integer, integer, mpc, result)
97   wrkpop (tasks, mpc, result)
98   gateway (tasks, mpc, result)
99   % Перебираем список задач, готовых к выполнению,
100  % для каждой вызываем wrkpush и выбираем наилучший
101  %результат
102  pktask (_, [], _, record (-1, _, _)) :- !.
103  pktask (tasklist, [record (task, tasktime, [])|tasktail],
104  procs, result) :-
105  wrkpush (tasklist, task, tasktime, procs, R1),
106  pktask (tasklist, tasktail, procs, R2),
107  pickless (R1, R2, result).
108  % Берем простаивающий процессор, назначаем ему задачу
109  % и либо дожидаемся выполнения % нескольких задач,
110  % либо берем следующую
111  wrkpush (tasklist, task, tasktime, record (work,
112  [record (stopTime, stopTasks)|stopTail]), result) :-
113  takeout (tasklist, task, newTasklist),
114  newTime = stopTime + taskTime,
115  desper (record (newTime, [tasks|stopTasks]), work, newWork),
116  wrkpop (newTasklist, record (newWork, stopTail), R1),
117  gateway (newTasklist, record (newWork, stopTail), R2),
118  pickless (R1, R2, result).
119  % Удаляем задачу из списка неназначенных, добавляем
120  % её в список назначенных этому процессору и перемещаем
121  % его в список работающих
122
123  % Дожидаемся выполнения нескольких задач
124  wrkpop ([], _, record (-1, [], [])) :- !.
125  wrkpop (tasklist, record ([], _), record (-1, [], [])) :- !.
126  % Признаем эту ветвь тупиковой, если все задачи назначены
127  % результат будет сформирован в gateway, или ни одна
128  % задача не выполняется
129  wrkpop (tasklist, record ([record (workTime,
130           [workTask|workTasktail])|workTail], stop), result) :-
131            remov (tasklist, workTask, newTasklist),
132  setStop (stop, record (workTime,
133           [workTask|workTasktail]), newStop),
134  wrkpop (newTasklist, record (workTail, newStop), R1),
135  gateway (newTasklist, record (workTail, newStop), R2),
136  pickless (R1, R2, result).
137  % Удаляем текущую задачу процессора из списка всех требований и
138  % перемещаем процессор в список простаивающих; далее либо
139  % дожидаемся выполнения ещё одной задачи, либо берём следующую
140
141  % Проверка возможности назначения следующей задачи, формирование
142  % списка задач, готовых к выполнению, формирование результата
143  gateway ([], record (work, stop),
144        record (length, work, stop)) :- tempo (work, length), !.
```

```
145  % Формирование результата
146  gateway (tasklist, record (_, []), record (-1, [], [])) :- !.
147  % Нет процессов, готовых взять задачу
148  gateway (tasklist, _, record (-1, [], [])) :-
149  deboss (tasklist, Q),
150  Q=[], !.
151  % Нет задач, готовых к выполнению
152  gateway (tasklist, procs, result) :-
153  deboss (tasklist, readyTasklist),
154  pktask (tasklist, readyTasklist, procs, result).
155  % Продолжение основного цикла
156  % Формирование списка из I процессоров
157  base (integer, procs)
158
159  base (0, []) :- !.
160  base (I, [record (0, [])|Q]) :-
161  I1 = I - 1,
162  base (I1, Q).
163
164  %Запускающая функция
165  start (tasks, integer, result)
166
167  start (Q, I, W) :-
168  base (I, R),
169  gateway (Q, record ([], R), W).
```

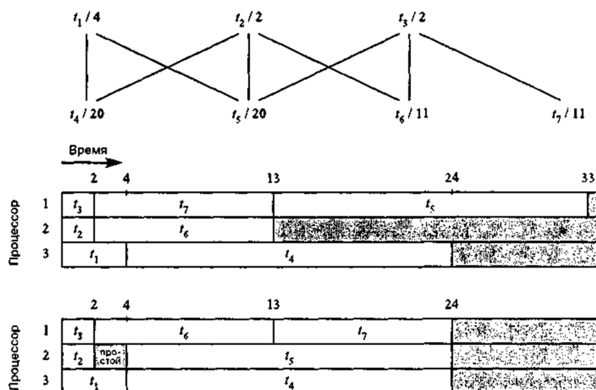

Рисунок 20. Результат планирования семи заданий на три процессора.

Одним из оптимальных решений, сформированных с помощью поиска по заданному критерию в проблемной области, определённой таким образом, является оптимальное расписание, показанное на рисунке 20. В верхней части рисунка показано отношение предшествования и про-

должительность заданий. В нижней — два варианта расписаний: не оптимальное с длительностью 33 и оптимальное с длительностью 24.

Оценка сложности. Задача относится к классу прямых переборных задач, решаемых с помощью алгоритма A^* тогда сложность составит $O(n \times k!)$, где n — число процессов, а k — число процессоров. На каждом шаге итерации хранятся четыре переменные на каждый узел дерева. Таким образом, рост объёма использованной памяти составит $O(2^l)$, где l — глубина рекурсии.

13. Литература

1. Братко, И. Программирование на языке Пролог для искусственного интеллекта / Братко И. ; под ред. А. М. Степанов; пер. с англ. А. И. Лупенко, А. М. Степанов .— Москва : Мир, 1990 .— 559 с.

2. Братко, Иван. Алгоритмы искусственного интеллекта на языке PROLOG : [пер. с англ.] / И. Братко ; Люблянский университет. Факультет компьютерных наук и информатики; Институт Йожефа Штефана .— 3-е изд .— М. [и др.] : Вильямс, 2004 .— 637 с.

3. Стобо, Джон. Язык программирования Пролог / Дж. Стобо ; пер. с англ. Н. Г. Волченкова, С. Г. Григорьева; под ред. Н. Г. Волченкова .— Москва : Радио и связь, 1993 .— 368 с.

4. Макаллистер, Дж. Искусственный интеллект и Пролог на микро-ЭВМ / Дж. Макаллистер ; пер. с англ. А. В. Чукашова, М. В. Сергиевского; под ред. М. В. Сергиевского .— Москва : Машиностроение, 1990 .— 235, [2] с.

5. Малпас, Джон. Реляционный язык Пролог и его применение / Дж. Малпас ; пер. с англ. А. А. Титова; под ред. В. Н. Соболева .— Москва : Наука : Гл. ред. физ.-мат. лит., 1990 .— 463 с.