**Реализация гамильтонова пути на Prolog**

Рисунок 1 программа и результат работы

Код программы

% Определяем граф в виде ребер

edge(0, 1).

edge(0, 2).

edge(0, 3).

edge(1, 0).

edge(1, 3).

edge(1, 4).

edge(2, 0).

edge(2, 3).

edge(3, 0).

edge(3, 1).

edge(3, 2).

edge(3, 4).

edge(4, 1).

edge(4, 3).

% Предикат для проверки, является ли путь гамильтоновым

hamiltonian\_path(Path) :-

length(Path, Length),

Length =< 5, % Задаем максимальное количество вершин в графе

Path = [Start|\_],

Start = 0, % Начинаем с первой вершины

hamiltonian\_path\_helper(Path, [Start]).

hamiltonian\_path\_helper([\_], \_). % Если осталась одна вершина, путь найден

hamiltonian\_path\_helper([Current, Next|Rest], Visited) :-

edge(Current, Next), % Проверяем наличие ребра между Current и Next

\+ member(Next, Visited), % Убедимся, что Next еще не посещен

hamiltonian\_path\_helper([Next|Rest], [Next|Visited]). % Рекурсивно продолжаем

% Для поиска пути, вызовите предикат hamiltonian\_path с нужной длиной

find\_hamiltonian\_path(Path) :-

length(Path, Length),

Length =< 5, % Задаем максимальное количество вершин в графе

hamiltonian\_path(Path).

Код запроса

find\_hamiltonian\_path(Path).

Ответы на вопросы

**1. Алгоритм «Британского музея»**

Алгоритм «Британского музея» используется для поиска гамильтонова пути в графе. Он основан на методе полного перебора, где исследуются все возможные пути. Эффективность этого алгоритма невысока, поскольку его сложность составляет O(n!)*O*(*n*!), где n*n* — количество вершин в графе. Это делает его непрактичным для больших графов.

**2. Генератор данных на проверку**

Генератор данных может быть реализован с помощью предикатов, которые создают случайные графы или наборы данных для тестирования. Вот пример генератора случайных графов:

% Генерация случайного графа

generate\_random\_graph(N, Edges) :-

findall((X, Y), (between(0, N-1, X), between(0, N-1, Y), X \= Y), AllEdges),

random\_permutation(AllEdges, ShuffledEdges),

length(Edges, N),

append(Edges, \_, ShuffledEdges).

**3. Общая схема алгоритма решения диофантова уравнения**

Диофантово уравнение имеет вид ax+by=c*ax*+*by*=*c*, где a*a*, b*b*, и c*c* — целые числа. Общая схема решения:

1. Проверка, существует ли решение (проверка на делимость).
2. Нахождение одного решения с использованием расширенного алгоритма Евклида.
3. Генерация всех решений, используя общее решение.

Пример реализации на Prolog:

% Расширенный алгоритм Евклида

extended\_gcd(A, 0, A, 1, 0).

extended\_gcd(A, B, G, X1, Y1) :-

B > 0,

extended\_gcd(B, A mod B, G, Y1, X1),

X is X1 - (A // B) \* Y1,

Y is X1.

% Поиск решения диофантова уравнения

diophantine(A, B, C, X, Y) :-

extended\_gcd(A, B, G, X0, Y0),

C mod G =:= 0, % Проверка на существование решения

X is X0 \* (C // G),

Y is Y0 \* (C // G).

**4. Общая математическая постановка задач на удовлетворение ограничений**

Задачи на удовлетворение ограничений (ЗУО) формулируются следующим образом:

* **Переменные**: Набор переменных, которые нужно определить.
* **Домены**: Для каждой переменной задается множество допустимых значений.
* **Ограничения**: Условия, которым должны удовлетворять значения переменных.

Пример реализации предиката для проверки ограничений:

% Пример задачи на удовлетворение ограничений

satisfy\_constraints(X, Y) :-

X + Y =:= 10,

X > 0,

Y > 0.

**5. Генератор чисел от 1 до 100 без использования списка значений**

Генератор чисел можно реализовать с помощью рекурсии:

% Генератор чисел от 1 до 100

generate\_numbers(101). % Базовый случай, когда достигли 101, завершаем

generate\_numbers(N) :-

N < 101,

write(N), nl, % Печатаем число

Next is N + 1,

generate\_numbers(Next). % Рекурсивный вызов для следующего числа

% Запуск генератора

start :-

generate\_numbers(1).

**Пример использования**

Чтобы запустить генератор чисел от 1 до 100, просто выполните запрос:

?- start.