Задание 1

% comb(+K, +List, -Comb) - построение сочетаний по k из элементов списка

comb(0, \_, []).

comb(K, [H|T], [H|CombTail]) :- K > 0, K1 is K - 1, comb(K1, T, CombTail).

comb(K, [\_|T], Comb) :- K > 0, comb(K, T, Comb).

% all\_combs\_up\_to\_k(+List, +K, -AllCombs) - построение всех сочетаний

% от 1 до k из элементов списка

all\_combs\_up\_to\_k(List, K, AllCombs) :- findall(Comb, (between(1, K, N), comb(N, List, Comb)), AllCombs).

% list\_length(+list, -L, +CurL) - подсчёт длины списка

% (надо для выставления максимального k в all\_combs\_up\_to\_k)

list\_length([], L, L) :- !.

list\_length([\_|T], L, CurL) :- NewL is CurL + 1, list\_length(T, L, NewL).

% volume\_sum(+List, -Sum, +CurV) - сумма объёмов

volume\_sum([], Sum, Sum) :- !.

volume\_sum([[V, \_]|T], Sum, CurV) :- NewV is CurV + V, volume\_sum(T, Sum, NewV).

% weight\_sum(+List, -Sum, +CurW) - сумма веса

weight\_sum([], Sum, Sum) :- !.

weight\_sum([[\_, W]|T], Sum, CurW) :- NewW is CurW + W, weight\_sum(T, Sum, NewW).

% list\_prohod1(+List, +GivenV) - предикат для работы с 1 частью задания

% (нахождение подмножества, сумма объёмов которого равна заданному объёму)

list\_prohod1([], \_, \_) :- write("Volume cannot be reached"), fail, !.

list\_prohod1([H|T], GivenV) :- volume\_sum(H, Sum, 0), (Sum == GivenV -> write(H), nl, !; list\_prohod1(T, GivenV)).

% list\_prohod2(+List, +GivenV, -F, +CurF) - предикат для работы со 2

% частью задания

% (нахождение подмножества, сумма объёмов которого равна заданному объёму,

% а сумма весов минимальна)

list\_prohod2([], \_, F, F) :- !.

list\_prohod2([H|T], GivenV, F, CurF) :- volume\_sum(H, Sum, 0),

(Sum == GivenV -> append(CurF, [H], NewF); append(CurF,[],NewF)), list\_prohod2(T, GivenV, F, NewF).

% min\_weight(+List, -Min, +CurMin, +CurMinW) - нахождение сочетания с

% минимальным весом

% CurMin - текущее подмножество с минимальным весом

% CurMinW - текущий минимум веса

min\_weight([], Min, Min, \_) :- !.

min\_weight([H|T], Min, CurMin, CurMinW) :- weight\_sum(H, Sum, 0), (Sum < CurMinW -> NewMinW is Sum, append(H,[],NewMin);

NewMinW is CurMinW, append(CurMin,[],NewMin)), min\_weight(T, Min, NewMin, NewMinW).

backpack\_call(V, List) :- list\_length(List, Len, 0), all\_combs\_up\_to\_k(List, Len, AllCombs),

list\_prohod1(AllCombs, V), list\_prohod2(AllCombs, V, Found, []), write(Found), nl, min\_weight(Found, Min, [], 10000), write(Min), nl.

Конечно, давайте разберём каждый предикат более подробно.

**comb/3: Построение Сочетаний**

prolog

Копировать код

% comb(+K, +List, -Comb)

comb(0, \_, []). % Базовый случай: если K равно 0, то сочетание пустое.

comb(K, [H|T], [H|CombTail]) :-

K > 0, % Убедимся, что K положительно.

K1 is K - 1, % Уменьшим K на 1.

comb(K1, T, CombTail). % Рекурсивно вызываем comb для оставшегося списка.

comb(K, [\_|T], Comb) :-

K > 0, % Убедимся, что K положительно.

comb(K, T, Comb). % Пропустим текущий элемент и рекурсивно вызываем comb для оставшегося списка.

Этот предикат используется для построения всех сочетаний длины K из заданного списка. Он рекурсивно выбирает элементы из списка и собирает их в сочетание. Первый правило базируется на том, что при K = 0 сочетание пустое. Второе правило выбирает головной элемент списка и рекурсивно формирует сочетание для оставшегося списка. Третье правило пропускает головной элемент списка и также рекурсивно вызывает себя для оставшегося списка.

**all\_combs\_up\_to\_k/3: Построение всех Сочетаний до K**

prolog

Копировать код

% all\_combs\_up\_to\_k(+List, +K, -AllCombs)

all\_combs\_up\_to\_k(List, K, AllCombs) :-

findall(Comb, (between(1, K, N), comb(N, List, Comb)), AllCombs).

Этот предикат создаёт все сочетания от 1 до K элементов из заданного списка. Он использует findall, чтобы собрать все сочетания, вызывая предикат comb для каждого числа от 1 до K.

**list\_length/3: Работа с длиной списка**

prolog

Копировать код

% list\_length(+List, -Length, +CurrentLength)

list\_length([], Length, Length) :- !. % Базовый случай: если список пуст, то длина равна текущей длине.

list\_length([\_|T], Length, CurrentLength) :-

NewLength is CurrentLength + 1, % Увеличиваем текущую длину на 1.

list\_length(T, Length, NewLength). % Рекурсивно вызываем list\_length для оставшегося списка.

Этот предикат вычисляет длину заданного списка. Он рекурсивно проходит по списку, увеличивая текущую длину на 1 на каждом шаге.

**volume\_sum/3 и weight\_sum/3: Работа с объёмом и весом**

prolog

Копировать код

% volume\_sum(+List, -Sum, +CurrentVolume)

volume\_sum([], Sum, Sum) :- !. % Базовый случай: если список пуст, то сумма объёмов равна текущей сумме.

volume\_sum([[V, \_]|T], Sum, CurrentVolume) :-

NewVolume is CurrentVolume + V, % Увеличиваем текущий объём на V.

volume\_sum(T, Sum, NewVolume). % Рекурсивно вызываем volume\_sum для оставшегося списка.

% weight\_sum(+List, -Sum, +CurrentWeight)

weight\_sum([], Sum, Sum) :- !. % Базовый случай: если список пуст, то сумма весов равна текущей сумме.

weight\_sum([[\_, W]|T], Sum, CurrentWeight) :-

NewWeight is CurrentWeight + W, % Увеличиваем текущий вес на W.

weight\_sum(T, Sum, NewWeight). % Рекурсивно вызываем weight\_sum для оставшегося списка.

Эти предикаты вычисляют сумму объёмов и весов для списка пар [объём, вес]. Они рекурсивно проходят по списку и добавляют объём или вес каждого элемента к текущему значению.

**list\_prohod1/2 и list\_prohod2/4: Нахождение подмножества с заданным объёмом и минимальным весом**

prolog

Копировать код

% list\_prohod1(+List, +GivenVolume)

list\_prohod1([], \_, \_) :- write("Volume cannot be reached"), fail, !. % Базовый случай: если список пуст, выводим сообщение об ошибке.

list\_prohod1([H|T], GivenVolume) :-

volume\_sum(H, Sum, 0), % Вычисляем сумму объёмов текущего подмножества.

(Sum == GivenVolume -> write(H), nl, !; list\_prohod1(T, GivenVolume)). % Если сумма равна заданному объёму, выводим подмножество.

% list\_prohod2(+List, +GivenVolume, -Found, +CurrentFound)

list\_prohod2([], \_, Found, Found) :- !. % Базовый случай: если список пуст, возвращаем найденное подмножество.

list\_prohod2([H|T], GivenVolume, Found, CurrentFound) :-

volume\_sum(H, Sum, 0), % Вычисляем сумму объёмов текущего подмножества.

(Sum == GivenVolume -> append(CurrentFound, [H], NewFound); append(CurrentFound,[],NewFound)), % Если сумма равна заданному объёму, добавляем подмножество к найденному.

list\_prohod2(T, GivenVolume, Found, NewFound). % Рекурсивно вызываем list\_prohod2 для оставшегося списка.

Эти предикаты находят подмножества списка сумма объёмов которых равна заданному объёму. list\_prohod1/2 выводит первое найденное подмножество, а list\_prohod2/4 находит подмножество с минимальной суммой весов.

**min\_weight/4: Нахождение подмножества с минимальным весом**

prolog

Копировать код

% min\_weight(+List, -Min, +CurrentMin, +CurrentMinWeight)

min\_weight([], Min, Min, \_) :- !. % Базовый случай: если список пуст, возвращаем текущее минимальное подмножество.

min\_weight([H|T], Min, CurrentMin, CurrentMinWeight) :-

weight\_sum(H, Sum, 0), % Вычисляем сумму весов текущего подмножества.

(Sum < CurrentMinWeight -> NewMinWeight is Sum, append(H,[],NewMin);

NewMinWeight is CurrentMinWeight, append(CurrentMin,[],NewMin)), % Если сумма весов меньше текущего минимального, обновляем минимальное подмножество.

min\_weight(T, Min, NewMin, NewMinWeight). % Рекурсивно вызываем min\_weight для оставшегося списка.

Этот предикат находит подмножество списка с минимальной суммой весов.

**backpack\_call/2: Главный предикат**

prolog

Копировать код

backpack\_call(V, List) :-

list\_length(List, Length, 0), % Получаем длину списка.

all\_combs\_up\_to\_k(List, Length, AllCombs), % Получаем все сочетания списка.

list\_prohod1(AllCombs, V), % Находим подмножество с заданным объёмом.

list\_prohod2(AllCombs, V, Found, []), % Находим подмножество с заданным объёмом и минимальным весом.

write(Found), nl, % Выводим найденное подмножество.

min\_weight(Found, Min, [], 10000), % Находим подмножество с минимальным весом.

write(Min), nl. % Выводим найденное подмножество.

Этот предикат запускает алгоритм для решения задачи о рюкзаке. Он вычисляет длину списка, создаёт все сочетания от 1 до длины списка, находит подмножество с заданным объёмом, затем находит подмножество с заданным объёмом и минимальным весом, выводит найденные подмножества.

Задание 2

% Предикат для поиска произвольного максимального паросочетания в графе

maximal\_matching(Graph, Matching) :-

    findall(Edge, edge(Edge), Edges), % Получаем список всех рёбер в графе

    find\_matching(Edges, [], Matching).

% Предикат для поиска паросочетания

find\_matching([], Acc, Acc).

find\_matching([Edge|RestEdges], Acc, Matching) :-

    edge(Edge),

    \+ member(Edge, Acc),

    \+ conflicting(Edge, Acc),

    find\_matching(RestEdges, [Edge|Acc], Matching).

% Предикат для проверки конфликтующих рёбер

conflicting((X1, Y1), Matching) :-

    member((X2, Y2), Matching),

    (X1 = X2 ; X1 = Y2 ; Y1 = X2 ; Y1 = Y2).

% Определение рёбер в графе

edge((a, b)).

edge((b, c)).

edge((c, d)).

edge((d, e)).

edge((a, c)).

edge((b, d)).

% Вызов предиката для поиска максимального паросочетания

?- maximal\_matching(Graph, Matching).

Этот код представляет собой реализацию предиката для поиска произвольного максимального паросочетания в графе. Давайте разберём каждую часть кода подробнее:

**Предикат maximal\_matching/2**

prolog

Копировать код

maximal\_matching(Graph, Matching) :-

findall(Edge, edge(Edge), Edges), % Получаем список всех рёбер в графе

find\_matching(Edges, [], Matching).

Этот предикат принимает граф Graph и возвращает произвольное максимальное паросочетание Matching. Сначала он использует предикат findall/3, чтобы собрать список всех рёбер в графе, используя предикат edge/1. Затем он вызывает предикат find\_matching/3, передавая список рёбер и пустой аккумулятор.

**Предикат find\_matching/3**

prolog

Копировать код

find\_matching([], Acc, Acc).

find\_matching([Edge|RestEdges], Acc, Matching) :-

edge(Edge),

\+ member(Edge, Acc),

\+ conflicting(Edge, Acc),

find\_matching(RestEdges, [Edge|Acc], Matching).

Этот предикат используется для поиска паросочетания. Он рекурсивно перебирает список рёбер. Если ребро Edge не конфликтует с уже выбранными рёбрами в Acc, оно добавляется к Acc. Затем предикат вызывается рекурсивно для оставшихся рёбер.

**Предикат conflicting/2**

prolog

Копировать код

conflicting((X1, Y1), Matching) :-

member((X2, Y2), Matching),

(X1 = X2 ; X1 = Y2 ; Y1 = X2 ; Y1 = Y2).

Этот предикат используется для проверки конфликтующих рёбер в паросочетании. Он проверяет, что ребро (X1, Y1) не имеет общих вершин с каким-либо ребром из Matching.

**Определение рёбер в графе**

prolog

Копировать код

edge((a, b)).

edge((b, c)).

edge((c, d)).

edge((d, e)).

edge((a, c)).

edge((b, d)).

Это определение рёбер в графе. В этом примере граф представлен как набор пар вершин, образующих рёбра.

3.5

\+ member(Edge, Acc) что значит эта запись

ChatGPT

Выражение \+ member(Edge, Acc) в Prolog означает отрицание (отрицание посылки) предиката member(Edge, Acc). Другими словами, оно проверяет, что элемент Edge **не** является членом списка Acc.

В контексте данного кода, где Acc представляет собой список выбранных рёбер, \+ member(Edge, Acc) используется, чтобы убедиться, что текущее ребро Edge ещё не было выбрано в паросочетание. Если ребро уже есть в списке Acc, оно не будет добавлено в паросочетание.

3.5

распиши мой код максимально подробно, с комментариями для каждой строчки

ChatGPT

Конечно, давайте разберём ваш код более подробно, добавив комментарии к каждой строке:

prolog

Копировать код

% Предикат для поиска произвольного максимального паросочетания в графе

maximal\_matching(Graph, Matching) :-

findall(Edge, edge(Edge), Edges), % Получаем список всех рёбер в графе

find\_matching(Edges, [], Matching).

% Предикат для поиска паросочетания

find\_matching([], Acc, Acc). % Базовый случай: если список рёбер пуст, возвращаем текущее паросочетание

find\_matching([Edge|RestEdges], Acc, Matching) :-

edge(Edge), % Проверяем, является ли ребро допустимым

\+ member(Edge, Acc), % Убеждаемся, что ребро ещё не включено в текущее паросочетание

\+ conflicting(Edge, Acc), % Проверяем, что ребро не конфликтует с уже выбранными рёбрами

find\_matching(RestEdges, [Edge|Acc], Matching). % Рекурсивно вызываем предикат для оставшихся рёбер

% Предикат для проверки конфликтующих рёбер

conflicting((X1, Y1), Matching) :-

member((X2, Y2), Matching), % Проверяем каждое ребро из паросочетания

(X1 = X2 ; X1 = Y2 ; Y1 = X2 ; Y1 = Y2). % Проверяем наличие общих вершин

% Определение рёбер в графе

edge((a, b)).

edge((b, c)).

edge((c, d)).

edge((d, e)).

edge((a, c)).

edge((b, d)).

Надеюсь, это поможет вам лучше понять ваш код! Если у вас возникнут дополнительные вопросы или потребуется дополнительная помощь, не стесняйтесь спрашивать.

Этот код на языке Prolog предназначен для поиска произвольного максимального паросочетания в графе. Паросочетание в графе — это множество рёбер, таких что никакие два рёбра не имеют общих вершин.

**Предикат maximal\_matching/2**

prolog

Копировать код

maximal\_matching(Graph, Matching) :-

findall(Edge, edge(Edge), Edges), % Получаем список всех рёбер в графе

find\_matching(Edges, [], Matching).

1. maximal\_matching/2 — основной предикат, который находит максимальное паросочетание в графе.
2. findall(Edge, edge(Edge), Edges) — создаёт список всех рёбер в графе. edge(Edge) собирает все рёбра, определённые в графе, а findall/3 сохраняет их в список Edges.
3. find\_matching(Edges, [], Matching) — вызывает предикат find\_matching/3 для поиска паросочетания. Edges — это список всех рёбер, [] — начальный пустой список для накопления рёбер паросочетания, Matching — переменная, в которую будет сохранено найденное паросочетание.

**Предикат find\_matching/3**

prolog

Копировать код

find\_matching([], Acc, Acc).

find\_matching([Edge|RestEdges], Acc, Matching) :-

edge(Edge),

\+ member(Edge, Acc),

\+ conflicting(Edge, Acc),

find\_matching(RestEdges, [Edge|Acc], Matching).

1. find\_matching([], Acc, Acc) — рекурсивное окончание: если список рёбер пуст, накопленный список (Acc) является результатом.
2. find\_matching([Edge|RestEdges], Acc, Matching) — основная рекурсивная часть, которая обрабатывает рёбра по одному:
   * edge(Edge) — проверяет, что Edge действительно является ребром в графе.
   * \+ member(Edge, Acc) — проверяет, что Edge ещё не добавлено в накопленный список Acc.
   * \+ conflicting(Edge, Acc) — проверяет, что Edge не конфликтует с уже добавленными рёбрами в Acc.
   * find\_matching(RestEdges, [Edge|Acc], Matching) — если все проверки прошли, добавляет Edge в накопленный список и продолжает обработку оставшихся рёбер (RestEdges).

**Предикат conflicting/2**

prolog

Копировать код

conflicting((X1, Y1), Matching) :-

member((X2, Y2), Matching),

(X1 = X2 ; X1 = Y2 ; Y1 = X2 ; Y1 = Y2).

1. conflicting((X1, Y1), Matching) — проверяет, что ребро (X1, Y1) конфликтует с любым ребром в паросочетании Matching.
2. member((X2, Y2), Matching) — перебирает все рёбра (X2, Y2) в паросочетании Matching.
3. (X1 = X2 ; X1 = Y2 ; Y1 = X2 ; Y1 = Y2) — проверяет, есть ли общие вершины между ребром (X1, Y1) и каким-либо ребром (X2, Y2) из Matching.

**Определение рёбер в графе**

prolog

Копировать код

edge((a, b)).

edge((b, c)).

edge((c, d)).

edge((d, e)).

edge((a, c)).

edge((b, d)).

Здесь определены рёбра графа, которые используются в предикате edge/1.

**Итог**

Этот код рекурсивно строит паросочетание, проверяя для каждого ребра, можно ли его добавить в текущее паросочетание, не создавая конфликтов с уже добавленными рёбрами. Результат — максимальное паросочетание, хранящееся в переменной Matching.

приведи мне пример паросочетания

ChatGPT

Конечно, давайте рассмотрим пример паросочетания для заданного графа. Мы рассмотрим процесс построения паросочетания с использованием данного кода.

Вот список рёбер нашего графа:

prolog

Копировать код

edge((a, b)).

edge((b, c)).

edge((c, d)).

edge((d, e)).

edge((a, c)).

edge((b, d)).

Теперь рассмотрим процесс нахождения паросочетания с использованием предиката find\_matching/3.

**Пример построения паросочетания:**

1. Начинаем с пустого списка паросочетания: [].
2. Проверяем первое ребро (a, b):
   * Оно не конфликтует с пустым списком, поэтому добавляем его: [(a, b)].
3. Проверяем второе ребро (b, c):
   * Оно конфликтует с уже добавленным (a, b) (общая вершина b), поэтому пропускаем его.
4. Проверяем третье ребро (c, d):
   * Оно не конфликтует с текущим паросочетанием [(a, b)], поэтому добавляем его: [(a, b), (c, d)].
5. Проверяем четвёртое ребро (d, e):
   * Оно конфликтует с уже добавленным (c, d) (общая вершина d), поэтому пропускаем его.
6. Проверяем пятое ребро (a, c):
   * Оно конфликтует с уже добавленным (a, b) (общая вершина a), поэтому пропускаем его.
7. Проверяем шестое ребро (b, d):
   * Оно конфликтует с уже добавленным (a, b) (общая вершина b), поэтому пропускаем его.

Итак, мы получили паросочетание [(a, b), (c, d)].

**Другой возможный пример паросочетания:**

Процесс может быть различным, так как порядок обработки рёбер влияет на конечный результат. Рассмотрим другой порядок:

1. Начинаем с пустого списка паросочетания: [].
2. Проверяем первое ребро (b, c):
   * Оно не конфликтует с пустым списком, поэтому добавляем его: [(b, c)].
3. Проверяем второе ребро (a, b):
   * Оно конфликтует с уже добавленным (b, c) (общая вершина b), поэтому пропускаем его.
4. Проверяем третье ребро (c, d):
   * Оно конфликтует с уже добавленным (b, c) (общая вершина c), поэтому пропускаем его.
5. Проверяем четвёртое ребро (d, e):
   * Оно не конфликтует с текущим паросочетанием [(b, c)], поэтому добавляем его: [(b, c), (d, e)].
6. Проверяем пятое ребро (a, c):
   * Оно конфликтует с уже добавленным (b, c) (общая вершина c), поэтому пропускаем его.
7. Проверяем шестое ребро (b, d):
   * Оно конфликтует с уже добавленным (b, c) (общая вершина b), поэтому пропускаем его.

Итак, мы получили другое паросочетание [(b, c), (d, e)].

Таким образом, в зависимости от порядка обработки рёбер, паросочетания могут отличаться, но оба примера являются корректными паросочетаниями.

этот код подходит для ориентированного или неориентированного графа?

ChatGPT

Данный код подходит для неориентированного графа. В неориентированном графе рёбра не имеют направления, то есть ребро между вершинами a и b может быть записано как (a, b) или (b, a), и это одно и то же ребро.

**Поддержка неориентированного графа**

Вот несколько причин, почему данный код предназначен для неориентированного графа:

1. **Определение рёбер:** Рёбра определены в виде пар (X, Y), где порядок вершин не имеет значения. В неориентированном графе ребро (a, b) эквивалентно ребру (b, a), но в нашем коде мы рассматриваем их как одно и то же ребро.
2. **Проверка конфликтов:** Предикат conflicting/2 проверяет, пересекаются ли вершины рёбер. Он делает это, проверяя четыре возможных случая:

prolog

Копировать код

(X1 = X2 ; X1 = Y2 ; Y1 = X2 ; Y1 = Y2)

Это условие верно для неориентированного графа, так как порядок вершин не имеет значения.

**Поддержка ориентированного графа**

Для ориентированного графа (где рёбра имеют направление) необходимо внести изменения в код, так как направление рёбер важно, и (a, b) не эквивалентно (b, a).

1. **Определение рёбер:** Рёбра должны быть определены с учётом направления, например:

prolog

Копировать код

edge((a, b)). % Ребро от a к b

edge((b, c)). % Ребро от b к c

1. **Проверка конфликтов:** Конфликты в ориентированном графе должны учитывать направление рёбер. Если направление важно, предикат conflicting/2 должен быть изменён так, чтобы учитывать только совпадение исходящей или входящей вершины, в зависимости от задачи.

**Пример изменений для ориентированного графа:**

prolog

Копировать код

% Предикат для поиска произвольного максимального паросочетания в ориентированном графе

maximal\_matching\_directed(Graph, Matching) :-

findall(Edge, edge(Edge), Edges), % Получаем список всех рёбер в графе

find\_matching\_directed(Edges, [], Matching).

% Предикат для поиска паросочетания

find\_matching\_directed([], Acc, Acc).

find\_matching\_directed([Edge|RestEdges], Acc, Matching) :-

edge(Edge),

\+ member(Edge, Acc),

\+ conflicting\_directed(Edge, Acc),

find\_matching\_directed(RestEdges, [Edge|Acc], Matching).

% Предикат для проверки конфликтующих рёбер в ориентированном графе

conflicting\_directed((X1, Y1), Matching) :-

member((X2, Y2), Matching),

(X1 = X2 ; Y1 = Y2).

% Предикат для поиска произвольного максимального паросочетания в графе

maximal\_matching(Graph, Matching) :-

findall(Edge, edge(Edge), Edges), % Получаем список всех рёбер в графе

find\_matching(Edges, [], Matching).

% Предикат для поиска паросочетания

find\_matching([], Acc, Acc).

find\_matching([Edge|RestEdges], Acc, Matching) :-

edge(Edge),

\+ member(Edge, Acc),

\+ conflicting(Edge, Acc),

find\_matching(RestEdges, [Edge|Acc], Matching).

find\_matching([Edge|RestEdges], Acc, Matching):-

find\_matching(RestEdges, Acc, Matching).

% Предикат для проверки конфликтующих рёбер

conflicting((X1, Y1), Matching) :-

member((X2, Y2), Matching),

(X1 = X2 ; X1 = Y2 ; Y1 = X2 ; Y1 = Y2).

% Определение рёбер в графе

edge((a, b)).

edge((b, c)).

edge((c, d)).

edge((d, e)).

edge((a, c)).

edge((b, d)).

Задание 4

% Предикат для чтения строк из файла

read\_lines(FileName, Lines) :-

    open(FileName, read, Stream),

    read\_lines\_from\_stream(Stream, Lines),

    close(Stream).

read\_lines\_from\_stream(Stream, []) :-

    at\_end\_of\_stream(Stream).

read\_lines\_from\_stream(Stream, [Line|Rest]) :-

    \+ at\_end\_of\_stream(Stream),

    read\_line\_to\_string(Stream, Line),

    read\_lines\_from\_stream(Stream, Rest).

% Предикат для разделения строки на слова

split\_string\_on\_whitespace(String, Words) :-

    split\_string(String, " \t\n", "", Words).

% Предикат для удаления повторяющихся слов из списка

remove\_duplicates([], []).

remove\_duplicates([X|Xs], Ys) :-

    member(X, Xs),

    remove\_duplicates(Xs, Ys).

remove\_duplicates([X|Xs], [X|Ys]) :-

    \+ member(X, Xs),

    remove\_duplicates(Xs, Ys).

% Предикат для записи списка строк в файл

write\_lines\_to\_file(FileName, Lines) :-

    open(FileName, write, Stream),

    maplist(write\_line\_to\_stream(Stream), Lines),

    close(Stream).

write\_line\_to\_stream(Stream, Line) :-

    write(Stream, Line), nl(Stream).

% Основной предикат для выполнения задачи

main(InputFile, OutputFile) :-

    read\_lines(InputFile, InputLines),

    maplist(split\_string\_on\_whitespace, InputLines, WordsList),

    flatten(WordsList, AllWords),

    remove\_duplicates(AllWords, UniqueWords),

    maplist(atomics\_to\_string(' ',), UniqueWords, UniqueLines),

    write\_lines\_to\_file(OutputFile, UniqueLines).

Этот код на SWI-Prolog представляет собой программу, которая читает строки из файла, а затем выводит в отдельный файл только те строки, которые состоят из уникальных слов. Давайте разберём его по частям:

**Предикат read\_lines/2**

prolog

Копировать код

% Предикат для чтения строк из файла

read\_lines(FileName, Lines) :-

open(FileName, read, Stream),

read\_lines\_from\_stream(Stream, Lines),

close(Stream).

Этот предикат открывает файл для чтения, читает строки из него и возвращает список строк.

**Предикат read\_lines\_from\_stream/2**

prolog

Копировать код

read\_lines\_from\_stream(Stream, []) :-

at\_end\_of\_stream(Stream).

read\_lines\_from\_stream(Stream, [Line|Rest]) :-

\+ at\_end\_of\_stream(Stream),

read\_line\_to\_string(Stream, Line),

read\_lines\_from\_stream(Stream, Rest).

Этот предикат читает строки из потока до тех пор, пока не достигнет конца файла (at\_end\_of\_stream). Он рекурсивно добавляет каждую строку в список строк Rest.

**Предикат split\_string\_on\_whitespace/2**

prolog

Копировать код

split\_string\_on\_whitespace(String, Words) :-

split\_string(String, " \t\n", "", Words).

Этот предикат разделяет строку на слова, используя пробелы, символы табуляции и новой строки в качестве разделителей.

**Предикат remove\_duplicates/2**

prolog

Копировать код

remove\_duplicates([], []).

remove\_duplicates([X|Xs], Ys) :-

member(X, Xs),

remove\_duplicates(Xs, Ys).

remove\_duplicates([X|Xs], [X|Ys]) :-

\+ member(X, Xs),

remove\_duplicates(Xs, Ys).

Этот предикат удаляет повторяющиеся элементы из списка. Он рекурсивно проходит по списку и добавляет элементы в новый список, пропуская повторяющиеся.

**Предикат write\_lines\_to\_file/2**

prolog

Копировать код

write\_lines\_to\_file(FileName, Lines) :-

open(FileName, write, Stream),

maplist(write\_line\_to\_stream(Stream), Lines),

close(Stream).

Этот предикат записывает список строк в файл. Он открывает файл для записи, записывает каждую строку из списка с помощью предиката write\_line\_to\_stream/2 и закрывает файл после окончания записи.

**Предикат write\_line\_to\_stream/2**

prolog

Копировать код

write\_line\_to\_stream(Stream, Line) :-

write(Stream, Line), nl(Stream).

Этот предикат записывает строку в поток и добавляет символ новой строки (nl(Stream)) для перехода на новую строку.

**Предикат main/2**

prolog

Копировать код

main(InputFile, OutputFile) :-

read\_lines(InputFile, InputLines),

maplist(split\_string\_on\_whitespace, InputLines, WordsList),

flatten(WordsList, AllWords),

remove\_duplicates(AllWords, UniqueWords),

maplist(atomics\_to\_string(' ',), UniqueWords, UniqueLines),

write\_lines\_to\_file(OutputFile, UniqueLines).

Этот предикат представляет основную часть программы. Он читает строки из входного файла, разбивает их на слова, удаляет повторяющиеся слова, а затем записывает уникальные строки в выходной файл.

3.5

распиши мой код максимально подробно, с комментариями для каждой строчки

ChatGPT

Конечно, давайте разберем ваш код по строкам и добавим комментарии к каждой строке:

prolog

Копировать код

% Предикат для чтения строк из файла

read\_lines(FileName, Lines) :-

open(FileName, read, Stream), % Открываем файл для чтения и создаем поток

read\_lines\_from\_stream(Stream, Lines), % Вызываем вспомогательный предикат для чтения строк из потока

close(Stream). % Закрываем поток после чтения всех строк

% Базовый случай: если поток достиг конца, список строк пуст

read\_lines\_from\_stream(Stream, []) :-

at\_end\_of\_stream(Stream).

% Рекурсивный случай: если поток еще не достиг конца, считываем следующую строку и добавляем ее в список строк

read\_lines\_from\_stream(Stream, [Line|Rest]) :-

\+ at\_end\_of\_stream(Stream), % Проверяем, что поток еще не достиг конца

read\_line\_to\_string(Stream, Line), % Считываем строку из потока

read\_lines\_from\_stream(Stream, Rest). % Рекурсивно вызываем этот же предикат для оставшихся строк

% Предикат для разделения строки на слова

split\_string\_on\_whitespace(String, Words) :-

split\_string(String, " \t\n", "", Words). % Разделяем строку по пробелам, символам табуляции и новой строки и получаем список слов

% Предикат для удаления повторяющихся слов из списка

remove\_duplicates([], []). % Базовый случай: если список пуст, возвращаем пустой список

remove\_duplicates([X|Xs], Ys) :-

member(X, Xs), % Если голова списка встречается в хвосте списка...

remove\_duplicates(Xs, Ys). % ...то рекурсивно вызываем этот же предикат для хвоста

remove\_duplicates([X|Xs], [X|Ys]) :-

\+ member(X, Xs), % Если голова списка не встречается в хвосте списка...

remove\_duplicates(Xs, Ys). % ...то добавляем ее в результирующий список и рекурсивно вызываем этот же предикат для хвоста

% Предикат для записи списка строк в файл

write\_lines\_to\_file(FileName, Lines) :-

open(FileName, write, Stream), % Открываем файл для записи и создаем поток

maplist(write\_line\_to\_stream(Stream), Lines), % Записываем каждую строку из списка в файл

close(Stream). % Закрываем поток после записи всех строк

% Предикат для записи строки в поток

write\_line\_to\_stream(Stream, Line) :-

write(Stream, Line), nl(Stream). % Записываем строку в поток и добавляем символ новой строки

% Основной предикат для выполнения задачи

main(InputFile, OutputFile) :-

read\_lines(InputFile, InputLines), % Читаем строки из входного файла

maplist(split\_string\_on\_whitespace, InputLines, WordsList), % Разделяем каждую строку на слова

flatten(WordsList, AllWords), % Преобразуем список списков слов в один список слов

remove\_duplicates(AllWords, UniqueWords), % Удаляем повторяющиеся слова

maplist(atomics\_to\_string(' '), UniqueWords, UniqueLines), % Преобразуем каждое слово в строку

write\_lines\_to\_file(OutputFile, UniqueLines). % Записываем уникальные строки в выходной файл

Надеюсь, это поможет вам лучше понять ваш код! Если у вас возникнут дополнительные вопросы или потребуется дополнительная помощь, не стесняйтесь обращаться.

3.5

расскажи мне суть работы моего кода

ChatGPT

Конечно! Давайте разберемся, как работает ваш код.

1. **Чтение строк из файла**: Сначала ваш код открывает файл для чтения и создает поток (Stream). Затем он вызывает предикат read\_lines\_from\_stream/2, который рекурсивно читает строки из потока до тех пор, пока не достигнет конца файла. Каждая прочитанная строка добавляется в список Lines.
2. **Разделение строк на слова**: После того как строки из файла были прочитаны, ваш код применяет предикат split\_string\_on\_whitespace/2 к каждой строке, который разделяет строку на слова по пробелам, символам табуляции и новой строки.
3. **Удаление повторяющихся слов**: Затем ваш код применяет предикат remove\_duplicates/2 к списку слов, чтобы удалить повторяющиеся слова.
4. **Запись уникальных строк в файл**: После того как были получены уникальные слова, они преобразуются обратно в строки и записываются в выходной файл с помощью предиката write\_lines\_to\_file/2.

Таким образом, код выполняет следующие шаги: считывание строк из файла, разделение строк на слова, удаление повторяющихся слов и запись уникальных строк в выходной файл.

Задание 5

5. Номер 3797 обладает интересным свойством. Будучи простым, можно непрерывно удалять цифры слева направо и оставаться простыми на каждом этапе: 3797, 797, 97 и 7. Аналогично мы можем работать справа налево: 3797, 379, 37 и 3.

Найдите сумму простых чисел, меньших 1000000 которые можно обрезать слева направо и справа налево.

ПРИМЕЧАНИЕ. 2, 3, 5 и 7 не считаются усеченными простыми числами.

% prime(+N, +Del) - проверка числа на простоту

prime(N, Del) :- NewDel is Del + 1, Ost is N mod Del, (Ost == 0, Del > 1 ->

fail, !; (Del >= (N // 2) -> (N < 2 -> fail; true), !;

prime(N, NewDel))).

% checkNumLeft(+N) - проверка числа на соответствие условию при удалении

% справа налево

checkNumLeft(0) :- !.

checkNumLeft(N) :- NewN is N//10, (prime(N, 1) -> checkNumLeft(NewN); fail, !).

% checkNumRight(+N, -Pow, +RPow) - проверка числа на соответствие условию

% при удалении слева направо

% справа налево

checkNumRight(N, Pow, RPow) :- NewN is N mod Pow, (N == NewN -> RPow is Pow // 10, !;

NewPow is Pow \* 10, (prime(NewN, 1) -> checkNumRight(N, NewPow, RPow); fail, !)).

% sumChisl(+MaxN, +N, -Sum, +CurSum) - предикат прохода по числам от N до

% MaxN с проверкой на соответствие условию

% (громоздкий из-за большого числа условий для ускорения работы)

sumChisl(MaxN, MaxN, Sum, Sum) :- !.

sumChisl(MaxN, N, Sum, CurSum) :- NewN is N + 1, (checkNumRight(N, 10, Pow), FDigit is N // Pow,

(prime(FDigit, 1) -> checkNumLeft(N); fail) -> NewSum is CurSum + N, write(N), write(" is the needed number"), nl;

NewSum is CurSum), sumChisl(MaxN, NewN, Sum, NewSum).

call :- sumChisl(1000000, 10, Sum, 0), write(Sum).

Этот код на SWI-Prolog выполняет следующие действия:

1. **Проверка числа на простоту**:
   * Предикат prime/2 принимает на вход число N и делитель Del и проверяет, является ли число N простым. При этом проверяется, что число не делится на какое-либо число от 2 до N // 2.
2. **Проверка числа на соответствие условию при удалении справа налево**:
   * Предикат checkNumLeft/1 проверяет число на соответствие условию, при котором при удалении цифр справа налево результат также является простым числом.
3. **Проверка числа на соответствие условию при удалении слева направо справа налево**:
   * Предикат checkNumRight/3 проверяет число на соответствие условию, при котором при удалении цифр слева направо и справа налево результат также является простым числом.
4. **Предикат для подсчета суммы простых чисел**:
   * Предикат sumChisl/4 принимает на вход максимальное число MaxN, текущее число N, сумму Sum и текущую сумму CurSum. Он рекурсивно проходит от N до MaxN, проверяя каждое число на соответствие условиям и прибавляя к сумме те числа, которые удовлетворяют условиям.
5. **Вызов и вывод результата**:
   * Предикат call/0 запускает sumChisl/4 с максимальным числом 1000000 и начальными параметрами суммы 0. После завершения вычислений результат выводится на экран.

Данный код ищет и выводит сумму всех простых чисел меньше 1000000, которые при удалении цифр слева направо и справа налево также остаются простыми числами.

добавь в мой код комментарии с подробным пояснением

Конечно, вот ваш код с комментариями:

prolog

Копировать код

% prime(+N, +Del) - проверка числа на простоту

% Предикат для проверки, является ли число простым

prime(N, Del) :-

% Увеличиваем делитель на 1

NewDel is Del + 1,

% Вычисляем остаток от деления числа N на Del

Ost is N mod Del,

% Проверяем условие: если остаток равен 0 и делитель больше 1, то число не является простым

(Ost == 0, Del > 1 ->

% Если условие выполняется, возвращаем false

fail, !;

% Если условие не выполняется и делитель больше или равен половине числа, то число является простым

(Del >= (N // 2) ->

(N < 2 ->

fail;

true), !;

% Иначе рекурсивно вызываем предикат prime/2 с увеличенным на 1 делителем

prime(N, NewDel))).

% checkNumLeft(+N) - проверка числа на соответствие условию при удалении справа налево

% Проверяем, что число N при удалении цифр справа налево остается простым

checkNumLeft(0) :- !.

checkNumLeft(N) :-

% Получаем новое число при удалении последней цифры

NewN is N // 10,

% Если число N простое, проверяем новое число

(prime(N, 1) ->

checkNumLeft(NewN);

% Если число не простое, возвращаем false

fail, !).

% checkNumRight(+N, -Pow, +RPow) - проверка числа на соответствие условию при удалении слева направо справа налево

% Проверяем, что число N при удалении цифр слева направо и справа налево остается простым

checkNumRight(N, Pow, RPow) :-

% Получаем новое число при удалении первой цифры

NewN is N mod Pow,

% Если число N не изменилось, то оно простое и мы можем удалить первую цифру

(N == NewN ->

% Уменьшаем степень 10 в Pow на 1

RPow is Pow // 10,

!;

% Иначе увеличиваем степень 10 в Pow в 10 раз и проверяем новое число

NewPow is Pow \* 10,

% Если число простое, проверяем новое число

(prime(NewN, 1) ->

checkNumRight(N, NewPow, RPow);

% Если число не простое, возвращаем false

fail, !)).

% sumChisl(+MaxN, +N, -Sum, +CurSum) - предикат прохода по числам от N до MaxN с проверкой на соответствие условию

% (громоздкий из-за большого числа условий для ускорения работы)

% Подсчитываем сумму простых чисел, которые удовлетворяют условиям

sumChisl(MaxN, MaxN, Sum, Sum) :- !.

sumChisl(MaxN, N, Sum, CurSum) :-

% Увеличиваем число на 1

NewN is N + 1,

% Проверяем число на соответствие условиям удаления цифр слева направо и справа налево

(checkNumRight(N, 10, Pow),

% Получаем первую цифру числа

FDigit is N // Pow,

% Если первая цифра простое число, проверяем число при удалении цифр справа налево

(prime(FDigit, 1) ->

checkNumLeft(N);

% Если первая цифра не простое число, возвращаем false

fail) ->

% Если число удовлетворяет условиям, прибавляем его к сумме и выводим сообщение

NewSum is CurSum + N,

write(N), write(" is the needed number"), nl;

% Если число не удовлетворяет условиям, оставляем сумму неизменной

NewSum is CurSum),

% Рекурсивно вызываем предикат для следующего числа

sumChisl(MaxN, NewN, Sum, NewSum).

% call - запуск программы

% Запускаем программу, подсчитываем сумму и выводим результат

call :-

% Подсчитываем сумму простых чисел, которые удовлетворяют условиям

sumChisl(1000000, 10, Sum, 0),

% Выводим результат на экран

write(Sum).

В этих комментариях я подробно описал каждый предикат и указал, что делает каждая строка кода.

<https://disk.yandex.ru/d/wkt6QlmIj3DMoA> - ссылка на диск жука

<https://docs.google.com/spreadsheets/d/1WqnCvZGE3xGKorVyu8DZNraOBOEWw9FYLqbWk1wA_oI/edit#gid=1892006766> - ссылка на таблицу жука

<https://www.jetbrains.com/help/idea/settings-http-proxy.html> - ссылка на статью для настройки прокси

<https://docs.google.com/document/d/1nnLYOWT5FtapvESRJqg8jsN4-qHpWZapAw2ocFQ9Lkw/edit> - ссылка на документ

[proxy.kubsu.ru](https://vk.com/away.php?to=http%3A%2F%2Fproxy.kubsu.ru&cc_key=)

8080

<https://docs.google.com/document/d/1VpNXw5iE10mcuUa_-LIzAoJ738kVDrGGOrtvBti-Htw/edit>