UTF-8

M-arias y Araas

Lucía Jiang, b19m042

Table of Contents

code 1
Parte 1: Particiones M-arias de un número 1
Predicado 1.1 1
Predicado 1.2 1
Predicado 1.3 2
Parte 2: Ara^^c3^^b1as de expansión de un grafo 2
Predicado 2.1 2
Predicado 2.2 2
Tests 5
Usage and interface 6
Documentation on exports 6
pots/3 (pred) 6
pots_aux/4 (pred) 7
mpart/3 (pred) 7
mpart_aux/4 (pred) 7
maria/3 (pred) 7
arista/2 (pred) 7
guardar_grafo/1 (pred) 8
guardar_grafo_aux/1 (pred) 8
aranya/0 (pred) 8
arista_copia/2 (pred) 8
clonar_aristas/0 (pred) 8
clonar_aristas_aux/1 (pred) 8
lista_aristas_org/1 (pred) 9
lista_aristas/1 (pred) 9
num_aristas/1 (pred) 9
lista_vertices/1 (pred) 9
num_vertices/1 (pred) 9
num_aristas_a_quitar/1 (pred) 10
unido/2 (pred) 10
conexo/0 (pred) 10
conexo_aux/2 (pred) 10
existe_camino/2 (pred) 10
existe_camino_aux/3 (pred) 10
quitar_arista/0 (pred) 11
quitar_arista_aux/2 (pred) 11
eliminar/1 (pred) 11
grado_vertice/2 (pred) 11
un_unico_vertice_mas3/0 (pred) 12
un_unico_vertice_mas3_aux/2 (pred) 12
test_pots/3 (pred) 12
test_mpart/3 (pred) 14
test_maria/3 (pred) 18
test_guardar_grafo/1 (pred) 18
test_guardar_grafo_aux/1 (pred) 19
test_aranya/1 (pred) 19
Documentation on multifiles 20
^^Fcall_in_module/2 (pred) 20
Documentation on imports 20

References 21

code

Esta práctica consiste en repasar los conceptos fundamentales estudiados en el bloque temático de programación con ISO-Prolog. El trabajo se divide en 2 partes.

Parte 1: Particiones M-arias de un número

El objetivo de la primera es escribir un predicado `maria/3` tal que el tercer argumento es el número de particiones `M`-arias del segundo argumento siendo `M` el primer argumento.

Predicado 1.1

Dados `M` y `N` enteros, `Ps` es la lista con las potencias de `M` que son menores o iguales que `N` en orden descendiente.

```
pots(1,_1,[1]).
pots(M,N,Ps) :-
    M>1,
    pots_aux(M,N,L,1),
    reverse(L,Ps).
```

Para eso llamamos a la función auxiliar `pots_aux/4` que nos devuelve la lista que queremos en orden creciente, por lo que invertimos utilizando el predicado `reverse` predefinido.

`P` es la lista en orden creciente de las potencias de `M` menores que `N`. `Ins` se utiliza para tener un registro de cuál ha sido el último número introducido.

Terminamos el método cuando `Ins > N` y realizamos corte.

```
pots_aux(_1,N,[],Aux) :-
    Aux>N,
    !.
pots_aux(M,N,[Ins|Ps],Ins) :-
    Aux is Ins*M,
    pots_aux(M,N,Ps,Aux).
```

Predicado 1.2

Dados `M` y `N` enteros, devuelve `P` por backtracking todas las particiones `M`-arias de `N` representadas como listas de enteros. Para ello recurrimos a un método auxiliar.

```
mpart(M,N,P) :-
    mpart_aux(M,N,P,N).
```

El método auxiliar `mpart/4` se llama con los mismos parámetros que mpart, con un cuarto extra que almacena el último valor de la lista, de manera que los elementos de más a la derecha son siempre menores que los de su izquierda.

Obtenemos la lista `P` de las potencias menores que `M` utilizando el predicado definido en pots. Aprovechamos el método predefinido member para ir escogiendo los elementos de `P` con backtracking para mostrar todas las particiones `M`-arias.

```
mpart_aux(_1,0,[],_2).
mpart_aux(M,N,[X|P],Ant) :-
    pots(M,N,Ps),
    member(X,Ps),
    X=<N,
    X=<Ant,
    Sup is N-X,
    mpart_aux(M,Sup,P,X).
```

Predicado 1.3

`NPart` es el número de particiones `M`-arias de `N`.

Utilizaremos los predicados predefinidos `findall` para obtener en una lista todas las `M`-particiones posibles, y length para obtener el número `NPart`.

```
maria(M,N,NPart) :-
    findall(_1,mpart(M,N,_2),R),
    length(R,NPart).
```

Parte 2: Araas de expansión de un grafo

La segunda consiste en buscar si existen araas de expansión de un grafo. Una araa de expansión de un grafo es un árbol de expansión que además es una araa.

Un árbol de expansión es un subgrafo que es árbol y que contiene todos los vértices del grafo. Una araa es un grafo con como máximo un vértice cuyo grado (número de aristas incidentes a él) es 3 o más.

Por tanto, una araa de expansión de un grafo es un subgrafo que es un árbol, contiene todos los vértices del grafo, y como máximo un solo vértice de grado 3 o más.

Se asume que los grafos de entrada son conexos y no dirigidos, por lo que para representar la conexión entre dos vértices a,b, incluiremos en la base de hechos el hecho arista(a,b) o el hecho arista(b,a), pero no ambos.

Predicado 2.1

Dado un grafo representado como una lista de aristas `G`, se aserta en la base de datos como hechos del predicado `arista/2` los elementos de `G`.

Para ello, se define `arista/2` como predicado dinámico en el programa. Al llamar al predicado `guardar_grafo(G)` se borran todos los hechos que hubiera anteriormente.

Cuando se llame a este predicado, se hará un retractall de todos los predicados de la forma `arista(_,_)`, y se llama al predicado `guardar_grafo_aux/1` que aade los hechos recursivamente.

```
guardar_grafo(G) :-
    retractall(arista(_1,_2)),
    guardar_grafo_aux(G).
```

Se utiliza un predicado auxiliar, `guardar_grafo_aux/1` que recorre la lista `G` que contiene las aristas del grafo para aadir a la base de hechos. Se utiliza el predicado `assert` predefinido.

```
guardar_grafo_aux([]).
guardar_grafo_aux([G1|G]) :-
    assert(G1),
    guardar_grafo_aux(G).
```

Predicado 2.2

Sabemos que en un árbol `A+1=V`, siendo `A` el número de aristas y `V` el de vértices.

Además, para que sea araa hemos visto que tiene que ser conexo y con como máximo un vértice mayor que 3.

Para estudiar esto, definiremos varios predicados auxiliares previos:

- **Clonar las aristas de la base de hechos.** Todas las operaciones que realizaremos será en una base de hechos con los predicados `arista_copia`.

  Clonamos todas las `arista` de la base de hechos a otras, `arista_copia/2`, de manera que todas las modificaciones que realizamos son en esta posterior. Para ello, se recurre a una auxiliar, `clonar_aristas_aux/1` con la lista de aristas.

```
clonar_aristas :-
    retractall(arista_copia(_1,_2)),
    lista_aristas_org(L),
    clonar_aristas_aux(L).
```

  Además, utilizamos una auxiliar para recorrer la base de hechos, que encontramos en la lista `L` pasada por parámetro. Se inserta recursivamente en `arista_copia/2`, otro predicado dinámico.

```
clonar_aristas_aux([]).
clonar_aristas_aux([arista(X,Y)|L]) :-
    assert(arista_copia(X,Y)),
    clonar_aristas_aux(L).
```

- **Obtener número de vértices a quitar.** Hemos visto que en un árbol se cumple que `V=A+1`. Por tanto, miraremos cuántas aristas tenemos que quitar para que pueda ser un árbol, `N=A-V+1`. Cabe mencionar que esto es una condición necesaria para ser araa pero no suficiente.

```
num_aristas_a_quitar(N) :-
    num_aristas(A),
    num_vertices(V),
    N is A-V+1.
```

  - **Obtener listas de aristas**. Tendremos dos predicados, uno para las aristas originales y otra para las clonadas.

```
lista_aristas(L) :-
    findall(arista_copia(X,Y),arista_copia(X,Y),L).
```

```
lista_aristas_org(L) :-
    findall(arista(X,Y),arista(X,Y),L).
```

  - **Obtener lista de vértices.** Para no obtener repetidos, utilizamos la función sort, que nos deja un conjunto ordenado.

```
lista_vertices(Lord) :-
    findall(X,arista(X,_1),L1),
    findall(Y,arista(_2,Y),L2),
    append(L1,L2,L),
    sort(L,Lord).
```

  - **Obtener número de aristas**. Nos aprovechamos el predicado `lista_aristas/1` para calcular el número de aristas mirando la longitud de la lista.

```
num_aristas(N) :-
    findall(_1,arista(_2,_3),L),
    length(L,N).
```

  - **Obtener número de vértices**. Nos aprovechamos el predicado `lista_vertices/1` para calcular el número de vértices mirando la longitud de la lista.

```
num_vertices(N) :-
    lista_vertices(L),
    length(L,N).
```

- **Comprobar conexión.** Llamaremos a este predicado cada vez que quitemos una arista, pues no tiene sentido quitar una arista y que se desconecte el grafo, dejando de ser un árbol y por tanto araa.

Un grafo es conexo si dado cualquier vértice `V` del grafo, existe un camino entre `V` y cualquier vértice del grafo.

- **Dos vértices unidos.** El predicado es cierto si existe una arista de `X` a `Y` o una arista de `Y` a `X`. En definitiva, que estén unidos los dos vértices.

```
unido(X,X).
unido(X,Y) :-
    arista_copia(X,Y).
unido(X,Y) :-
    arista_copia(Y,X).
```

- **Existe camino entre dos vértices.** El predicado es cierto si existe un camino que lleve del vértice `X` a `Y`

```
existe_camino(X,Y) :-
    existe_camino_aux(X,Y,[X]),
    !.
```

Utilizamos un predicado auxiliar, `existe_camino_aux/3` que mira si existe un camino entre `X` y `Y` y almacena en `V` la lista de vértices que ya se han recorrido para no volver a acceder y evitar bucles infinitos.

```
existe_camino_aux(X,Y,_1) :-
    unido(X,Y).
existe_camino_aux(X,Y,V) :-
    unido(X,C),
    C\==Y,
    \+member(C,V),
    existe_camino_aux(C,Y,[C|V]).
```

- **Conexo.** El predicado es cierto si el grafo registrado en la base de datos es conexo.

```
conexo :-
    lista_vertices([L1|L]),
    conexo_aux(L1,L).
```

Con otra auxiliar y con la lista de vértices `V`, seleccionamos el primer vértice y buscamos si existe un camino que los una.

```
conexo_aux(_1,[]).
conexo_aux(X,[V1|V]) :-
    existe_camino(X,V1),
    conexo_aux(X,V).
```

- **Un único vértice con grado mayor que 3.**

Definiremos un predicado para obtener el grado de un vértice en particular.

```
grado_vertice(X,N) :-
    findall(X,arista_copia(X,_1),L1),
    findall(X,arista_copia(_2,X),L2),
    length(L1,N1),
    length(L2,N2),
    N is N1+N2.
```

Y posteriormente, evaluamos el grado de todos los vértices. En el caso de que más de un vértice diera con grado mayor que 3, salta un fallo.

```
un_unico_vertice_mas3 :-
    lista_vertices(L),
    un_unico_vertice_mas3_aux(L,0).
un_unico_vertice_mas3_aux([],N) :-
    !,
```

```
    N=<1.
un_unico_vertice_mas3_aux([L1|L],N) :-
    grado_vertice(L1,N1),
    N1>=3,
    !,
    N2 is N+1,
    un_unico_vertice_mas3_aux(L,N2).
un_unico_vertice_mas3_aux([_1|L],N) :-
    un_unico_vertice_mas3_aux(L,N).
```

- **Quitar aristas.** Utilizando los predicados declarados anteriormente, quitaremos las aristas que dejen al subgrafo conexo, y comprobaremos que el subgrafo final tenga como máximo un único vértice con grado 3 o más.

  Aprovechamos el predicado member para realizar estas operaciones con backtracking.

```
quitar_arista :-
    num_aristas_a_quitar(N),
    lista_aristas(L),
    quitar_arista_aux(N,L).
quitar_arista_aux(0,_1) :-
    !,
    conexo,
    un_unico_vertice_mas3.
quitar_arista_aux(N,L) :-
    conexo,
    member(X,L),
    call(X),
    eliminar(X),
    N2 is N-1,
    quitar_arista_aux(N2,L).
```

  Con el propósito de poder realizar backtracking correctamente, no podemos quitar simplemente las aristas de la base de hechos, pues en el caso de dar fallo y retornar, la arista se habría perdido. Por tanto, se aade una cláusula al predicado, que se accede si no existe la arista que se quiere retractar, a volver a aadirla y dar un fallo.

```
eliminar(A) :-
    retract(A).
eliminar(A) :-
    assert(A),
    fail.
```

Con todos estos predicados, podemos llamar al que se nos pedía: Dado un grafo `G` guardado en la base de datos, tiene éxito si el grafo proporcionado contiene una araa de expansión y falla finitamente en caso contrario.

```
aranya :-
    clonar_aristas,
    quitar_arista.
```

Tests

Con el propósito de testear los predicados, se han creado unos predicados cuyo objetivo es únicamente comprobar que todos funcionan de manera esperada.

- test_pots/3

```
test_pots(M,N,Ps) :-
    pots(M,N,Ps).
```

- test_mpart/3

```
test_mpart(M,N,Ps) :-
    mpart(M,N,Ps).
```

- test_maria/3

```
test_maria(M,N,NPart) :-
    maria(M,N,NPart).
```

- test_guardar_grafo/1

```
test_guardar_grafo(G) :-
    guardar_grafo(G),
    test_guardar_grafo_aux(G).
test_guardar_grafo_aux([]).
test_guardar_grafo_aux([G1|G]) :-
    call(G1),
    test_guardar_grafo_aux(G).
```

- test_aranya/1

```
test_aranya(G) :-
    guardar_grafo(G),
    aranya.
```

Usage and interface

- **Library usage:**
  `:- use_module(/Users/lucia/Desktop/ProgDecl/practica2/code.pl).`
- **Exports:**
  - *Predicates:*
    pots/3, pots_aux/4, mpart/3, mpart_aux/4, maria/3, arista/2, guardar_grafo/1, guardar_grafo_aux/1, aranya/0, arista_copia/2, clonar_aristas/0, clonar_aristas_aux/1, lista_aristas_org/1, lista_aristas/1, num_aristas/1, lista_vertices/1,
    num_vertices/1, num_aristas_a_quitar/1, unido/2, conexo/0, conexo_aux/2, existe_camino/2, existe_camino_aux/3, quitar_arista/0, quitar_arista_aux/2, eliminar/1, grado_vertice/2, un_unico_vertice_mas3/0, un_unico_vertice_mas3_aux/2, test_pots/3, test_mpart/3, test_maria/3, test_guardar_grafo/1, test_guardar_grafo_aux/1, test_aranya/1.
  - *Multifiles:*
    Σcall_in_module/2.

Documentation on exports

pots/3: PREDICATE

Usage: `pots(M,N,Ps)`

Dados `M` y `N` enteros, `Ps` es la lista con las potencias de `M` que son menores o iguales que `N` en orden descendiente.

```
pots(1,_1,[1]).
pots(M,N,Ps) :-
    M>1,
    pots_aux(M,N,L,1),
    reverse(L,Ps).
```

pots_aux/4: PREDICATE

Usage: `pots_aux(M,N,P,Ins)`

`P` es la lista en orden creciente de las potencias de `M` menores que `N`. `Ins` es el último valor introducido.

```
pots_aux(_1,N,[],Aux) :-
    Aux>N,
    !.
pots_aux(M,N,[Ins|Ps],Ins) :-
    Aux is Ins*M,
    pots_aux(M,N,Ps,Aux).
```

mpart/3: PREDICATE

Usage: `mpart(M,N,P)`

Dados `M` y `N` enteros, devuelve `P` por backtracking todas las particiones `M`-arias de `N` representadas como listas de enteros.

```
mpart(M,N,P) :-
    mpart_aux(M,N,P,N).
```

mpart_aux/4: PREDICATE

Usage: `mpart_aux(M,N,P,Ant)`

`P` es la lista de particiones `M`-arias de `N` representadas como listas de enteros.

```
mpart_aux(_1,0,[],_2).
mpart_aux(M,N,[X|P],Ant) :-
    pots(M,N,Ps),
    member(X,Ps),
    X=<N,
    X=<Ant,
    Sup is N-X,
    mpart_aux(M,Sup,P,X).
```

maria/3: PREDICATE

Usage: `maria(M,N,NPart)`

`NPart` es el número de particiones `M`-arias de `N`.

```
maria(M,N,NPart) :-
    findall(_1,mpart(M,N,_2),R),
    length(R,NPart).
```

arista/2: PREDICATE

Usage: `arista(X,Y)`

Predicado dinámico con el que definimos un grafo

The predicate is of type *dynamic*.

guardar_grafo/1: PREDICATE

Usage: `guardar_grafo(G)`

Dado un grafo representado como una lista de aristas `G`, se aserta en la base de datos como hechos del predicado arista/2 los elementos de `G`.

```
guardar_grafo(G) :-
    retractall(arista(_1,_2)),
    guardar_grafo_aux(G).
```

guardar_grafo_aux/1: PREDICATE

Usage: `guardar_grafo_aux(G)`

Recorre la lista `G` que contiene las aristas del grafo para aadir a la base de hechos.

```
guardar_grafo_aux([]).
guardar_grafo_aux([G1|G]) :-
    assert(G1),
    guardar_grafo_aux(G).
```

aranya/0: PREDICATE

Usage:

Dado un grafo `G` guardado en la base de datos, tiene éxito si el grafo proporcionado contiene una araa de expansión.

arista_copia/2: PREDICATE

Usage: `arista_copia(X,Y)`

Predicado dinámico que utilizamos para definir un grafo.

The predicate is of type *dynamic*.

clonar_aristas/0: PREDICATE

Usage:

Clona todas las aristas de la base de datos original a `aristas_copia/2`

```
clonar_aristas :-
    retractall(arista_copia(_1,_2)),
    lista_aristas_org(L),
    clonar_aristas_aux(L).
```

clonar_aristas_aux/1: PREDICATE

Usage: `clonar_aristas_aux(L)`

Clona todas las aristas de la base de datos original a `aristas_copia/2`

```
clonar_aristas_aux([]).
clonar_aristas_aux([arista(X,Y)|L]) :-
    assert(arista_copia(X,Y)),
    clonar_aristas_aux(L).
```

lista_aristas_org/1: PREDICATE

Usage: `lista_aristas_org(L)`

Obtenemos la lista `L` con todas las aristas que hay en la base de datos original.

```
lista_aristas_org(L) :-
    findall(arista(X,Y),arista(X,Y),L).
```

lista_aristas/1: PREDICATE

Usage: `lista_aristas(L)`

Obtenemos la lista `L` con todas las aristas que hay en la base de datos clonada.

```
lista_aristas(L) :-
    findall(arista_copia(X,Y),arista_copia(X,Y),L).
```

num_aristas/1: PREDICATE

Usage: `num_aristas(N)`

`N` es el número de aristas del grafo `G`.

```
num_aristas(N) :-
    findall(_1,arista(_2,_3),L),
    length(L,N).
```

lista_vertices/1: PREDICATE

Usage: `lista_vertices(L)`

`L` es la lista de los vértices del grafo.

```
lista_vertices(Lord) :-
    findall(X,arista(X,_1),L1),
    findall(Y,arista(_2,Y),L2),
    append(L1,L2,L),
    sort(L,Lord).
```

num_vertices/1: PREDICATE

Usage: `num_vertices(N)`

`N` es el número de vértices del grafo `G`.

```
num_vertices(N) :-
    lista_vertices(L),
    length(L,N).
```

num_aristas_a_quitar/1: PREDICATE

Usage: `num_aristas_a_quitar(N)`

`N=A-V+1`, `N` es el número de aristas que hay que quitar del grafo para que sea un árbol (condición necesaria pero no suficiente).

```
num_aristas_a_quitar(N) :-
    num_aristas(A),
    num_vertices(V),
    N is A-V+1.
```

unido/2: PREDICATE

Usage: `unido(X,Y)`

El predicado es cierto si existe una arista de `X` a `Y` o una arista de `Y` a `X`.

```
unido(X,X).
unido(X,Y) :-
    arista_copia(X,Y).
unido(X,Y) :-
    arista_copia(Y,X).
```

conexo/0: PREDICATE

Usage:

Cierto si el grafo de la base de datos es conexo.

```
conexo :-
    lista_vertices([L1|L]),
    conexo_aux(L1,L).
```

conexo_aux/2: PREDICATE

Usage: `conexo_aux(V,L)`

Cierto si para el vértice `V` existe un camino con todos los elementos de la lista `L`

```
conexo_aux(_1,[]).
conexo_aux(X,[V1|V]) :-
    existe_camino(X,V1),
    conexo_aux(X,V).
```

existe_camino/2: PREDICATE

Usage: `existe_camino(X,Y)`

El predicado es cierto si existe un camino que lleve del vértice `X` a `Y`

```
existe_camino(X,Y) :-
    existe_camino_aux(X,Y,[X]),
    !.
```

existe_camino_aux/3: PREDICATE

Usage: `existe_camino_aux(X,Y,V)`

Predicado auxiliar que mira si existe un camino entre `X` y `Y` y almacena en `V` la lista de vértices que ya se han recorrido.

```
existe_camino_aux(X,Y,_1) :-
    unido(X,Y).
existe_camino_aux(X,Y,V) :-
    unido(X,C),
    C\==Y,
    \+member(C,V),
    existe_camino_aux(C,Y,[C|V]).
```

quitar_arista/0: PREDICATE

Usage:

El predicado es cierto si se consiguen quitar `N` manteniendo que sea conexo.

```
quitar_arista :-
    num_aristas_a_quitar(N),
    lista_aristas(L),
    quitar_arista_aux(N,L).
```

quitar_arista_aux/2: PREDICATE

Usage: `quitar_arista_aux(N,L)`

Predicado auxiliar que elimina `N` aristas de la lista `L` manteniendo que sea conexo y que el grafo final resultante tenga solo un vértice con grado mayor que 3.

```
quitar_arista_aux(0,_1) :-
    !,
    conexo,
    un_unico_vertice_mas3.
quitar_arista_aux(N,L) :-
    conexo,
    member(X,L),
    call(X),
    eliminar(X),
    N2 is N-1,
    quitar_arista_aux(N2,L).
```

eliminar/1: PREDICATE

Usage: `eliminar(A)`

Predicado que elimina el predicado arista `A` de la base de hechos.

```
eliminar(A) :-
    retract(A).
eliminar(A) :-
    assert(A),
    fail.
```

grado_vertice/2: PREDICATE

Usage: `grado_vertice(X,N)`

`N` es el grado del vértice `X`.

```
grado_vertice(X,N) :-
    findall(X,arista_copia(X,_1),L1),
    findall(X,arista_copia(_2,X),L2),
    length(L1,N1),
    length(L2,N2),
    N is N1+N2.
```

un_unico_vertice_mas3/0: PREDICATE

Usage:

Predicado que es cierto si hay como máximo un vértice con grado 3 o más.

```
un_unico_vertice_mas3 :-
    lista_vertices(L),
    un_unico_vertice_mas3_aux(L,0).
```

un_unico_vertice_mas3_aux/2: PREDICATE

Usage: `un_unico_vertice_mas3_aux(L,N)`

Predicado auxiliar que dado la lista de vértices existente en el grafo `L`, determina si hay como máximo un vértice de grado 3 o más.

```
un_unico_vertice_mas3_aux([],N) :-
    !,
    N=<1.
un_unico_vertice_mas3_aux([L1|L],N) :-
    grado_vertice(L1,N1),
    N1>=3,
    !,
    N2 is N+1,
    un_unico_vertice_mas3_aux(L,N2).
un_unico_vertice_mas3_aux([_1|L],N) :-
    un_unico_vertice_mas3_aux(L,N).
```

test_pots/3: PREDICATE

Usage: `test_pots(M,N,Ps)`

Comprueba que los resultados del predicado sean los esperados.

```
test_pots(M,N,Ps) :-
    pots(M,N,Ps).
```

Other properties:

Test: `test_pots(M,N,Ps)`

– *If the following properties hold at call time:*

`M=2` (`= /2`)

`N=12` (`= /2`)

then the following properties should hold upon exit:

`Ps=[8,4,2,1]` (= /2)

then the following properties should hold globally:

All the calls of the form `test_pots(M,N,Ps)` do not fail. (`not_fails/1`)

Test: `test_pots(M,N,Ps)`

– *If the following properties hold at call time:*

`M=3` (= /2)

`N=50` (= /2)

then the following properties should hold upon exit:

`Ps=[27,9,3,1]` (= /2)

then the following properties should hold globally:

All the calls of the form `test_pots(M,N,Ps)` do not fail. (`not_fails/1`)

Test: `test_pots(M,N,Ps)`

– *If the following properties hold at call time:*

`M=4` (= /2)

`N=100` (= /2)

then the following properties should hold upon exit:

`Ps=[64,16,4,1]` (= /2)

then the following properties should hold globally:

All the calls of the form `test_pots(M,N,Ps)` do not fail. (`not_fails/1`)

Test: `test_pots(M,N,Ps)`

Falta el elemento 1

– *If the following properties hold at call time:*

`M=2` (= /2)

`N=12` (= /2)

`Ps=[8,4,2]` (= /2)

then the following properties should hold globally:

Calls of the form `test_pots(M,N,Ps)` fail. (`fails/1`)

Test: `test_pots(M,N,Ps)`

Lista en orden contrario

– *If the following properties hold at call time:*

`M=3` (= /2)

`N=50` (= /2)

`Ps=[1,3,9,27]` (= /2)

then the following properties should hold globally:

Calls of the form `test_pots(M,N,Ps)` fail. (`fails/1`)

Test: `test_pots(M,N,Ps)`

Se excede el límite superior

– *If the following properties hold at call time:*

`M=4` (= /2)

`N=50` (= /2)

`Ps=[64,16,4,1]` (= /2)

then the following properties should hold globally:

Calls of the form `test_pots(M,N,Ps)` fail. (`fails/1`)

test_mpart/3: PREDICATE

Usage: `test_mpart(M,N,Ps)`

Comprueba que los resultados del predicado sean los esperados.

```
test_mpart(M,N,Ps) :-
    mpart(M,N,Ps).
```

Other properties:

Test: `test_mpart(M,N,P)`

– *If the following properties hold at call time:*

`M=2` (= /2)

`N=12` (= /2)

`P=[8,4]` (= /2)

then the following properties should hold globally:

All the calls of the form `test_mpart(M,N,P)` do not fail. (`not_fails/1`)

Test: `test_mpart(M,N,P)`

– *If the following properties hold at call time:*

`M=2` (= /2)

`N=12` (= /2)

`P=[8,2,2]` (= /2)

then the following properties should hold globally:

All the calls of the form `test_mpart(M,N,P)` do not fail. (`not_fails/1`)

Test: `test_mpart(M,N,P)`

– *If the following properties hold at call time:*

`M=2` (= /2)

`N=12` (= /2)

`P=[8,2,1,1]` (= /2)

then the following properties should hold globally:

All the calls of the form `test_mpart(M,N,P)` do not fail. (`not_fails/1`)

Test: `test_mpart(M,N,P)`

– *If the following properties hold at call time:*

`M=2` (= /2)

`N=12` (= /2)

`P=[8,1,1,1,1]` (= /2)

then the following properties should hold globally:

All the calls of the form `test_mpart(M,N,P)` do not fail. (`not_fails/1`)

Test: `test_mpart(M,N,P)`

– *If the following properties hold at call time:*

`M=2` (= /2)

`N=12` (= /2)

`P=[4,4,4]` (= /2)

then the following properties should hold globally:

All the calls of the form `test_mpart(M,N,P)` do not fail. (`not_fails/1`)

Test: `test_mpart(M,N,P)`

– *If the following properties hold at call time:*
`M=2` (= /2)
`N=12` (= /2)
`P=[4,4,2,2]` (= /2)
then the following properties should hold globally:
All the calls of the form `test_mpart(M,N,P)` do not fail. (`not_fails/1`)

Test: `test_mpart(M,N,P)`
– *If the following properties hold at call time:*
`M=2` (= /2)
`N=12` (= /2)
`P=[4,4,2,1,1]` (= /2)
then the following properties should hold globally:
All the calls of the form `test_mpart(M,N,P)` do not fail. (`not_fails/1`)

Test: `test_mpart(M,N,P)`
– *If the following properties hold at call time:*
`M=2` (= /2)
`N=12` (= /2)
`P=[4,4,1,1,1,1]` (= /2)
then the following properties should hold globally:
All the calls of the form `test_mpart(M,N,P)` do not fail. (`not_fails/1`)

Test: `test_mpart(M,N,P)`
– *If the following properties hold at call time:*
`M=2` (= /2)
`N=12` (= /2)
`P=[4,2,2,2,2]` (= /2)
then the following properties should hold globally:
All the calls of the form `test_mpart(M,N,P)` do not fail. (`not_fails/1`)

Test: `test_mpart(M,N,P)`
– *If the following properties hold at call time:*
`M=2` (= /2)
`N=12` (= /2)
`P=[4,2,2,2,1,1]` (= /2)
then the following properties should hold globally:
All the calls of the form `test_mpart(M,N,P)` do not fail. (`not_fails/1`)

Test: `test_mpart(M,N,P)`
– *If the following properties hold at call time:*
`M=2` (= /2)
`N=12` (= /2)
`P=[4,2,2,1,1,1,1]` (= /2)
then the following properties should hold globally:
All the calls of the form `test_mpart(M,N,P)` do not fail. (`not_fails/1`)

Test: `test_mpart(M,N,P)`

– *If the following properties hold at call time:*
`M=2` (= /2)
`N=12` (= /2)
`P=[4,2,1,1,1,1,1,1]` (= /2)
then the following properties should hold globally:
All the calls of the form `test_mpart(M,N,P)` do not fail. (`not_fails/1`)

Test: `test_mpart(M,N,P)`
– *If the following properties hold at call time:*
`M=2` (= /2)
`N=12` (= /2)
`P=[4,1,1,1,1,1,1,1,1]` (= /2)
then the following properties should hold globally:
All the calls of the form `test_mpart(M,N,P)` do not fail. (`not_fails/1`)

Test: `test_mpart(M,N,P)`
– *If the following properties hold at call time:*
`M=2` (= /2)
`N=12` (= /2)
`P=[2,2,2,2,2,2]` (= /2)
then the following properties should hold globally:
All the calls of the form `test_mpart(M,N,P)` do not fail. (`not_fails/1`)

Test: `test_mpart(M,N,P)`
– *If the following properties hold at call time:*
`M=2` (= /2)
`N=12` (= /2)
`P=[2,2,2,2,2,1,1]` (= /2)
then the following properties should hold globally:
All the calls of the form `test_mpart(M,N,P)` do not fail. (`not_fails/1`)

Test: `test_mpart(M,N,P)`
– *If the following properties hold at call time:*
`M=2` (= /2)
`N=12` (= /2)
`P=[2,2,2,2,1,1,1,1]` (= /2)
then the following properties should hold globally:
All the calls of the form `test_mpart(M,N,P)` do not fail. (`not_fails/1`)

Test: `test_mpart(M,N,P)`
– *If the following properties hold at call time:*
`M=2` (= /2)
`N=12` (= /2)
`P=[2,2,2,1,1,1,1,1,1]` (= /2)
then the following properties should hold globally:
All the calls of the form `test_mpart(M,N,P)` do not fail. (`not_fails/1`)

Test: `test_mpart(M,N,P)`

– *If the following properties hold at call time:*
`M=2` (= /2)
`N=12` (= /2)
`P=[2,2,1,1,1,1,1,1,1,1]` (= /2)
then the following properties should hold globally:
All the calls of the form `test_mpart(M,N,P)` do not fail. (`not_fails/1`)

Test: `test_mpart(M,N,P)`
– *If the following properties hold at call time:*
`M=2` (= /2)
`N=12` (= /2)
`P=[2,1,1,1,1,1,1,1,1,1,1]` (= /2)
then the following properties should hold globally:
All the calls of the form `test_mpart(M,N,P)` do not fail. (`not_fails/1`)

Test: `test_mpart(M,N,P)`
– *If the following properties hold at call time:*
`M=2` (= /2)
`N=12` (= /2)
`P=[1,1,1,1,1,1,1,1,1,1,1,1]` (= /2)
then the following properties should hold globally:
All the calls of the form `test_mpart(M,N,P)` do not fail. (`not_fails/1`)

Test: `test_mpart(M,N,P)`
Las listas tienen que estar en orden decreciente
– *If the following properties hold at call time:*
`M=2` (= /2)
`N=12` (= /2)
`P=[8,1,2,1]` (= /2)
then the following properties should hold globally:
Calls of the form `test_mpart(M,N,P)` fail. (`fails/1`)

Test: `test_mpart(M,N,P)`
Las listas tienen que estar en orden decreciente
– *If the following properties hold at call time:*
`M=2` (= /2)
`N=12` (= /2)
`P=[4,8]` (= /2)
then the following properties should hold globally:
Calls of the form `test_mpart(M,N,P)` fail. (`fails/1`)

Test: `test_mpart(M,N,P)`
No suman N
– *If the following properties hold at call time:*
`M=2` (= /2)
`N=12` (= /2)
`P=[8,2]` (= /2)
then the following properties should hold globally:
Calls of the form `test_mpart(M,N,P)` fail. (`fails/1`)

test_maria/3: PREDICATE

Usage: `test_maria(M,N,NPart)`

Comprueba que los resultados del predicado sean los esperados.

```
test_maria(M,N,NPart) :-
    maria(M,N,NPart).
```

Other properties:

Test: `test_maria(M,N,NPart)`

– *If the following properties hold at call time:*

`M=2` (`= /2`)

`N=12` (`= /2`)

then the following properties should hold upon exit:

`NPart=20` (`= /2`)

then the following properties should hold globally:

All the calls of the form `test_maria(M,N,NPart)` do not fail. (`not_fails/1`)

Test: `test_maria(M,N,NPart)`

– *If the following properties hold at call time:*

`M=3` (`= /2`)

`N=48` (`= /2`)

then the following properties should hold upon exit:

`NPart=72` (`= /2`)

then the following properties should hold globally:

All the calls of the form `test_maria(M,N,NPart)` do not fail. (`not_fails/1`)

Test: `test_maria(M,N,NPart)`

– *If the following properties hold at call time:*

`M=4` (`= /2`)

`N=50` (`= /2`)

then the following properties should hold upon exit:

`NPart=28` (`= /2`)

then the following properties should hold globally:

All the calls of the form `test_maria(M,N,NPart)` do not fail. (`not_fails/1`)

Test: `test_maria(M,N,NPart)`

Número incorrecto

– *If the following properties hold at call time:*

`M=3` (`= /2`)

`N=9` (`= /2`)

`NPart=1` (`= /2`)

then the following properties should hold globally:

Calls of the form `test_maria(M,N,NPart)` fail. (`fails/1`)

test_guardar_grafo/1: PREDICATE

Usage: `test_guardar_grafo(G)`

Comprueba que se han guardado todos los hechos de la lista `G`

```
test_guardar_grafo(G) :-
    guardar_grafo(G),
    test_guardar_grafo_aux(G).
```

Other properties:

Test: `test_guardar_grafo(G)`

– *If the following properties hold at call time:*

`G=[arista(a,b),arista(a,c),arista(a,d),arista(e,a),arista(a,f),arista(a,g)]` (`= /2`)

then the following properties should hold globally:

All the calls of the form `test_guardar_grafo(G)` do not fail. (`not_fails/1`)

Test: `test_guardar_grafo(G)`

– *If the following properties hold at call time:*

`G=[arista(a,b),arista(a,d),arista(e,f),arista(a,e),arista(c,b)]` (`= /2`)

then the following properties should hold globally:

All the calls of the form `test_guardar_grafo(G)` do not fail. (`not_fails/1`)

Test: `test_guardar_grafo(G)`

– *If the following properties hold at call time:*

`G=[arista(a,b),arista(c,b),arista(x,y)]` (`= /2`)

then the following properties should hold globally:

All the calls of the form `test_guardar_grafo(G)` do not fail. (`not_fails/1`)

test_guardar_grafo_aux/1: PREDICATE

Usage: `test_guardar_grafo_aux(G)`

Auxiliar para comprobar que se han guardado todos los hechos de la lista `G`

```
test_guardar_grafo_aux([]).
test_guardar_grafo_aux([G1|G]) :-
    call(G1),
    test_guardar_grafo_aux(G).
```

test_aranya/1: PREDICATE

Usage: `test_aranya(G)`

Comprueba que el grafo `G` contiene una araa.

```
test_aranya(G) :-
    guardar_grafo(G),
    aranya.
```

Other properties:

Test: `test_aranya(G)`

– *If the following properties hold at call time:*

`G=[arista(a,b),arista(a,c),arista(a,d),arista(e,a),arista(a,f),arista(a,g)]` (`= /2`)

then the following properties should hold globally:

All the calls of the form `test_aranya(G)` do not fail. (`not_fails/1`)

Test: `test_aranya(G)`

– *If the following properties hold at call time:*

`G=[arista(a,b),arista(a,d),arista(e,f),arista(a,e),arista(c,b)]` (`= /2`)

then the following properties should hold globally:

All the calls of the form `test_aranya(G)` do not fail. (`not_fails/1`)

Test: `test_aranya(G)`

– *If the following properties hold at call time:*

`G=[arista(a,b),arista(a,c),arista(a,d),arista(a,e),arista(b,c),arista(c,e),arista` (`= /2`)

then the following properties should hold globally:

All the calls of the form `test_aranya(G)` do not fail. (`not_fails/1`)

Test: `test_aranya(G)`

Grafo no conexo

– *If the following properties hold at call time:*

`G=[arista(a,b),arista(c,b),arista(x,y)]` (`= /2`)

then the following properties should hold globally:

Calls of the form `test_aranya(G)` fail. (`fails/1`)

Test: `test_aranya(G)`

Más de un vértice con grado 3 o más

– *If the following properties hold at call time:*

`G=[arista(a,b),arista(a,c),arista(a,d),arista(b,e),arista(b,g),arista(e,f)]` (`= /2`)

then the following properties should hold globally:

Calls of the form `test_aranya(G)` fail. (`fails/1`)

Documentation on multifiles

Σcall_in_module/2: PREDICATE

No further documentation available for this predicate. The predicate is *multifile*.

Documentation on imports

This module has the following direct dependencies:

– *Application modules:*

`operators`, `dcg_phrase_rt`, `datafacts_rt`, `dynamic_rt`, `classic_predicates`.

– *Internal (engine) modules:*

`term_basic`, `arithmetic`, `atomic_basic`, `basiccontrol`, `exceptions`, `term_compare`, `term_typing`, `debugger_support`, `hiord_rt`, `stream_basic`, `io_basic`, `runtime_control`, `basic_props`.

– *Packages:*

`prelude`, `initial`, `condcomp`, `classic`, `runtime_ops`, `dcg`, `dcg/dcg_phrase`, `dynamic`, `datafacts`, `assertions`, `assertions/assertions_basic`, `regtypes`.

References

(this section is empty)