Biblioteka kombinatoryczno-grafowa dla programu Maxima

ver. 2.2, 8 lutego 2018 r.

Autor: Antoni Szczepański, aszczep@prz.edu.pl, PRz-WEiI-KEiPI-Rzeszow-Poland.

Lista ponad 200 zaimplementowanych funkcji, wraz z krótkim opisem:

is_permutation_by_cycles(p)	- sprawdza, czy p jest permutacją zapisaną w postaci złożenia rozłącznych cykli
is_permutation_by_row(p)	- sprawdza, czy p jest permutacją zapisaną w postaci 1-wierszowej
p_size(p)	- zwraca liczbę n, która określa, do jakiego S_n należy permutacja p
p_value(p, i)	- zwraca wartość permutacji p dla argumentu i, czyli zwraca p[i]
p_fixed_points(p)	- zwraca listę punktów stałych permutacji p
p_canonical_form(p)	- zapisuje permutację p w cyklowej postaci kanonicznej, tzn. cykle są posortowane rosnąco ze względu na najmniejszy element w każdym cyklu i te najmniejsze elementy znajdują się na początkach swoich cykli
p_rotate(p, i)	- cyklicznie przesuwa permutację p, w zapisie 1-wierszowym, o i elementów w prawo, gdy i jest dodatnie (o i elementów w lewo, gdy i jest ujemne)
p_inverse(p)	- wyznacza permutację odwrotną do p, czyli p^{-1}
p_reverse(p)	- zwraca permutację, której postać 1-wierszowa powstała z zapisania od końca postaci 1-wierszowej permutacji p
p_sign(p)	- wyznacza znak permutacji p
p_type(p)	- wyznacza typ permutacji p
p_order(p)	- wyznacza rząd permutacji p

---

p_composition(p1, p2)	- wyznacza złożenie permutacji p1 i p2: p1*p2
p_list_composition(lista)	- wyznacza złożenie „od prawej do lewej” wszystkich permutacji wymienionych na liście lista
p_random(n)	- zwraca losową permutację ze zbioru Sn
e(n)	- zwraca permutację identycznościową w Sn

c(p)	- wyznacza liczbę wszystkich cykli w permutacji p (z uwzględnieniem cykli długości 1, nawet tych pominiętych)
c(p, k)	- wyznacza liczbę cykli długości k w permutacji p (dla k=1 uwzględnia także pominięte cykle dł. 1)
cpd(p)	- wyznacza liczbę cykli parzystej długości w permutacji p
cnpd(p)	- wyznacza liczbę cykli nieparzystej długości w permutacji p (z uwzględnieniem cykli długości 1, nawet tych pominiętych)

perm2cycles(p)	- konwertuje permutację p z zapisu 1-wierszowego do zapisu cyklowego; jeżeli n (oznaczające Sn, do którego należy permutacja p), jest punktem stałym permutacji p, to w zapisie cyklowym będzie to jedyny cykl długości 1; jeżeli natomiast n nie jest punktem stałym permutacji p, wtedy wszystkie cykle długości 1 zostaną pominięte.
perm2mincycles(p)	- konwertuje permutację p, zapisaną 1-wierszowo lub cyklowo, do minimalnego zapisu cyklowego, tzn. pomija wszystkie punkty stałe (cykle długości 1). Ta funkcja powstała głównie w celach przejrzystego wypisywania permutacji. Użycie jej do innych celów jest niewskazane, gdyż jeżeli największy element (n) permutacji p jest punktem stałym, to pominięcie tego elementu obniża Sn, do którego należy permutacja p!
perm2maxcycles(p)	- konwertuje permutację p, zapisaną 1-wierszowo lub cyklowo, do pełnego zapisu cyklowego, tzn. takiego, w którym występują wszystkie cykle długości 1. Ta funkcja powstała głównie na potrzeby ładnego wypisywania listy permutacji. Standardowo należy używać funkcji perm2cycles(p), gdyż wszystkie funkcje biblioteki permutacyjnej działają szybciej, gdy pomija się punkty stałe w zapisie cyklowym.

cycles2perm(p)	- konwertuje permutację p z zapisu cyklowego do 1-wierszowego
perm2matrix(p)	- konwertuje permutację p do kwadratowej macierzy zero-jedynkowej,
matrix2perm(m)	- konwertuje zero-jedynkową macierz kwadratową m do permutacji w zapisie 1-wierszowym

---

p_power1(p, k)	- oblicza k-tą potęgę permutacji p, gdy k jest liczbą całkowitą (+,-,0)
p_power2(p, k)	- oblicza p do potęgi ułamkowej 1/k, czyli wyznacza wszystkie permutacje g, które spełniają równanie p = g^k
p_power3(p, num, den)	- oblicza p do potęgi ułamkowej num/den, czyli wyznacza wszystkie permutacje g, które spełniają równanie p^num = g^den
p_power(p, alfa)	- oblicza p do potęgi alfa; alfa może być liczbą całkowitą lub ułamkiem zwykłym
p_solve(k, g)	- rozwiązuje równanie f^k = g, czyli wyznacza wszystkie permutacje f, które podniesione do k-tej potęgi dają g
p_solve2(rownanie, n)	- rozwiązuje w Sn dowolne równanie permutacyjne z jedną niewiadomą. Równanie należy zdefiniować jako funkcję jednej zmiennej (niewiadomej permutacji). Do rozdzielenia lewej i prawej strony równania należy użyć symbolu ==. Zmienna funkcyjna (np. p) może występować w funkcji definiującej równanie dowolną liczbę razy, w dowolnej potędze całkowitej. Permutacje stałe można podawać 1-wierszowo lub cyklowo. Nawiasami () można wymusić kolejność operacji. Równanie w jednym Sn może mieć rozwiązania, a w innym nie. Należy pamiętać, że dla zapisu cyklowego permutacja należy do takiego Sn, ile wynosi największa liczba występująca w jej zapisie cyklowym. Przykład równania i jego rozwiązanie:

```
(%i71)  rownanie(p) := p##[[4,2,3]]##p@-1==([[3,1],[4,2]]@-3##p)@11##[1,3,2,4]$
        p_solve2(rownanie, 4);
(%o71)  [[2,1,3,4],[4,3,1,2]]
```

p_solve3(rownanie1, rownanie2, n) – rozwiązuje dowolny układ dwóch równań permutacyjnych z dwiema niewiadomymi w Sn, tzn. zwraca wszystkie pary permutacji p i q, które powodują, że oba równania są spełnione. Może istnieć wiele par permutacji będących rozwiązaniem danego układu równań. Oto przykład układu, który akurat posiada tylko jedno rozwiązanie:

```
(%i7)     rownanie1(p,q) := p##(q@-11)==[1,3,4,2]$
          rownanie2(p,q) := q@4==[4,2,3,1]@7##p$
          rozwiazanie : p_solve3(rownanie1,rownanie2, 4);
(rozwiazanie) [[[4,2,3,1],[4,3,1,2]]]
```

p_filter(warunki, n) – wyznacza te permutacje w Sn, które spełniają zadane warunki. Warunki definiuje się za pomocą funkcji boolowskiej jednej zmiennej, w której mogą wystąpić operatory and, or, xor i not. Nawiasami () można wymusić priorytet działań. Przykład:

```
(%i107)   warunki(p) := (not is_even(p)) and (evenp(p_order(p)) or is_derangement(p)) and c(p)<3$
          wynik : p_filter(warunki,4);
(wynik)   [[2,3,4,1],[2,4,1,3],[3,1,4,2],[3,4,2,1],[4,1,2,3],[4,3,1,2]]
```

---

is_derangement(p) – sprawdza, czy permutacja p jest nieporządkiem

is_involution(p) – sprawdza, czy permutacja p jest inwolucją

is_transposition(p) – sprawdza, czy permutacja p jest transpozycją

is_onecyclic(p) – sprawdza, czy permutacja p jest jednocyklowa

is_equal(p1, p2) – sprawdza, czy permutacje p1 i p2 są równe

is_even(p) – sprawdza, czy permutacja p jest parzysta

is_odd(p) – sprawdza, czy permutacja p jest nieparzysta

---

p_inv_vector(p) – zwraca wektor inwersyjny permutacji p

p_inv_vector2(nr, n) – zwraca wektor inwersyjny dla permutacji o numerze nr, w porządku leksykograficznym, w Sn

p_inversions(p) – zwraca listę inwersji permutacji p

p_inversions_count(p) – zwraca liczbę inwersji permutacji p

p_number(p) – zwraca numer permutacji p na liście permutacji w porządku leksykograficznym

p_previous(p) – zwraca permutację poprzedzającą p na liście w porządku leksykograficznym

p_next(p) - zwraca permutację następną po p na liście w porządku leksykograficznym

p_from_number(nr, n) - wyznacza permutację o zadanym numerze w Sn, na liście w porządku leksykograficznym

p_from_number_plain_changes1(nr, n) - zwraca permutację o numerze nr w Sn, na liście permutacji wygenerowanych algorytmem „minimum zmian sąsiednich elementów”, gdy po permutacji „wędruje” systematycznie liczba n

p_from_number_plain_changes2(nr, n) - zwraca permutację o numerze nr w Sn, na liście permutacji wygenerowanych algorytmem „minimum zmian sąsiednich elementów”, gdy po permutacji „wędruje” systematycznie liczba 1

---

is_proper_type(typ) - sprawdza, czy typ permutacji jest poprawnie zapisany jako lista par, np. typ $1^2 3^1 4^2$ powinien być zakodowany tak: [[1,2],[3,1],[4,2]].

p_type_size(typ) - oblicza liczbę permutacji danego typu

p_type_sign(typ) - zwraca znak typu typ, tzn. znak dowolnej permutacji typu typ

p_type_order(typ) - zwraca rząd typu typ, tzn. rząd dowolnej permutacji typu typ

p_type_power(typ, k) - zwraca typ, jaki powstanie po podniesieniu typu typ do k-tej potęgi

p_all_types_of_order(n, rz) - zwraca wszystkie te typy permutacji w Sn, które posiadają rząd rz

p_all_types(n) - zwraca listę wszystkich typów permutacji w Sn

p_all_orders(n) - zwraca listę wszystkich możliwych rzędów w Sn

---

p_transpos_neighb_left(p) - przedstawia permutację p w postaci złożenia transpozycji sąsiednich elementów, przy sprowadzaniu elementów w kolejności 1, 2, ..., n, na swoje naturalne pozycje, czyli do lewej strony

p_transpos_neighb_right(p) - przedstawia permutację p w postaci złożenia transpozycji sąsiednich elementów, przy sprowadzaniu elementów w kolejności n, n-1, ..., 1, na swoje naturalne pozycje, czyli do prawej strony

p_transpos_neighb_by_order(p, f) – próbuje(!) przedstawić permutację p w postaci złożenia transpozycji sąsiednich elementów, sprowadzając elementy permutacji p na swoje naturalne pozycje w kolejności podanej w permutacji f; określenie „próbuje” oznacza, że często otrzymana lista transpozycji po złożeniu nie daje oryginalnej permutacji p(!), ponieważ w wyniku sprowadzania elementów permutacji p nie powstaje permutacja identycznościowa

p_transpos1(p) - przedstawia permutację p w postaci złożenia transpozycji, stosując do każdego cyklu długości k (k>2) regułę:
(n1,n2,n3,...,nk) = (n1,nk)(n1,n(k-1))
...(n1,n3)(n1,n2) <-- k-1 transpozycji

p_transpos2(p) - przedstawia permutację p w postaci złożenia transpozycji, stosując do każdego cyklu długości k (k>2) regułę:
(n1,n2,n3,...,nk) = (n1,n2)(n2,n3)(n3,n4)
...(n(k-1),nk) <-- k-1 transpozycji

p_to_involutions_comp(p) - przedstawia permutację p w postaci złożenia dwóch inwolucji na wszystkie możliwe sposoby

p_to_transpositions_comp(p,[m]) - przedstawia permutację p w postaci złożenia minimalnej liczby transpozycji (niekoniecznie sąsiednich elementów) na wszystkie możliwe sposoby. Opcjonalna liczba całkowita m wymusza liczbę transpozycji.

p_to_transpositions_neighb_comp(p,[m]) - przedstawia permutację p w postaci złożenia minimalnej liczby transpozycji sąsiednich elementów na wszystkie możliwe sposoby (minimalna liczba transpozycji jest równa liczbie inwersji permutacji p). Opcjonalna liczba całkowita m wymusza liczbę transpozycji.

---

p_all(n) - zwraca listę wszystkich permutacji w Sn, w porządku leksykograficznym, których jest n!

p_all_plain_changes1(n) - zwraca listę wszystkich permutacji w Sn, wygenerowanych algorytmem „minimum zmian

	sąsiednich elementów”, gdy po permutacji „wędruje” systematycznie liczba n; jest ich n!
p_all_plain_changes2(n)	- zwraca listę wszystkich permutacji w Sn, wygenerowanych algorytmem „minimum zmian sąsiednich elementów”, gdy po permutacji „wędruje” systematycznie liczba 1; jest ich n!
p_all_involutions(n)	- zwraca listę wszystkich inwolucji w Sn, których jest sum(n!/((n-2*k)!*2^k*k!), k, 0, floor(n/2))
p_all_derangements(n)	- zwraca listę wszystkich nieporządków w Sn, których jest n!*sum((-1)^k/k!, k, 0, n); lub inaczej a(0)= 1, a(n)= floor(n!/e + 1/2) dla n > 0 , gdzie e=2.718281828
p_all_partial_derangements(n, k)	– zwraca listę wszystkich nieporządków częściowych, czyli tych permutacji w Sn, które posiadają dokładnie k punktów stałych
p_all_transpositions(n)	- zwraca listę wszystkich transpozycji w Sn, których jest n*(n-1)/2
p_all_onecyclic(n)	- zwraca listę wszystkich permutacji jedno-cyklowych w Sn, których jest (n-1)!
p_all_k_cyclic(n, k)	- zwraca listę wszystkich permutacji k-cyklowych w Sn
p_all_with_k_inversions(n, k)	- zwraca listę tych wszystkich permutacji w Sn, które posiadają k inwersji
p_all_even(n)	- zwraca listę wszystkich permutacji parzystych w Sn, których jest n!/2
p_all_odd(n)	- zwraca listę wszystkich permutacji nieparzystych w Sn, których jest n!/2
p_all_of_type(typ)	- zwraca wszystkie permutacje typu typ
p_all_of_order(n, rzad)	- zwraca wszystkie permutacje rzędu rzad w Sn,
p_all_from_to(nr1, nr2, n)	- zwraca listę permutacji w Sn o numerach w po-rządku leksykograficznym od nr1 do nr2 włącznie; zwraca (nr2-nr1)+1 permutacji
p_all_multiset_perm(lista)	- generuje wszystkie permutacje (z powtórzeniami) zbioru z powtórzeniami (multizbioru), zapisanego jako lista, zawierającego liczby naturalne

---

is_group(lista) - sprawdza, czy podana lista permutacji jest grupą permutacji; zwraca true, jeżeli tak jest; w przeciwnym wypadku zwraca błąd (powód, dla którego lista nie jest grupą permutacji). To sprawdzenie dla licznej listy zajmuje trochę czasu, dlatego żadna inna funkcja, w której argumentem jest grupa permutacji, nie sprawdza, czy lista jest grupą. Takie rozwiązanie przyjęto z tego powodu, że zazwyczaj raz sprawdza się, czy dana lista permutacji jest grupą, a następnie, jeśli jest, wykonuje się na tej grupie permutacji różne obliczenia, poprzez wywołanie kilku funkcji. Sprawdzenie w każdej funkcji, czy lista permutacji jest grupą, zajęłoby dużo czasu.

is_commutative_group(grupa) - funkcja sprawdza, czy grupa permutacji jest przemienna (abelowa).

p_group_facts(grupa) - funkcja zwraca listę par permutacji, które są wzajemnie odwrotne w podanej grupie oraz zwraca tablicę mnożenia tej grupy; dla skrócenia zapisu każda permutacja jest zakodowana liczbą określającą jej pozycję na liście *grupa* (licząc od 1).

p_group_values(grupa, x) - funkcja zwraca listę wartości wszystkich permutacji grupy dla argumentu x.

p_fix_x_subgroup(grupa, x) - funkcja zwraca te permutacje grupy, które fiksują element x, czyli w których x jest punktem stałym. Permutacje te stanowią podgrupę grupy.

---

p_group_identity(n) - zwraca grupę identycznościową, tzn. grupę złożoną z jednej permutacji (identycznościowej e(n))

p_group_cyclic(n) - zwraca listę n permutacji, które stanowią grupę cykliczną Cn. Są to zakodowane permutacjami wierzchołków obroty n-kąta foremnego wokół jego „przyszpilonego” środka, przeprowadzające za każdym razem ten n-kąt na siebie.

p_group_dihedral(n) - zwraca listę 2n permutacji, które tworzą grupę dihedralną (grupę dwuścianu). Tę grupę stanowią zakodowane permutacjami wierzchołków obroty n-kąta foremnego względem jego „przyszpilonego”

środka, których jest n, oraz symetrie osiowe odwracające wielokąt „na drugą” stronę, których też jest n. Określenie *dwuścian* wynika z faktu, że n-kąt ma dwie strony (dwie ściany) i można go odwracać. Każda permutacja z grupy dwuścianu przeprowadza wierzchołki n-kąta na siebie nawzajem.

p_group_alternating(n) - zwraca listę n!/2 permutacji parzystych w Sn, które tworzą tzw. grupę alternującą

p_group_symmetric(n) - zwraca listę n! permutacji w Sn, które tworzą tzw. grupę symetryczną

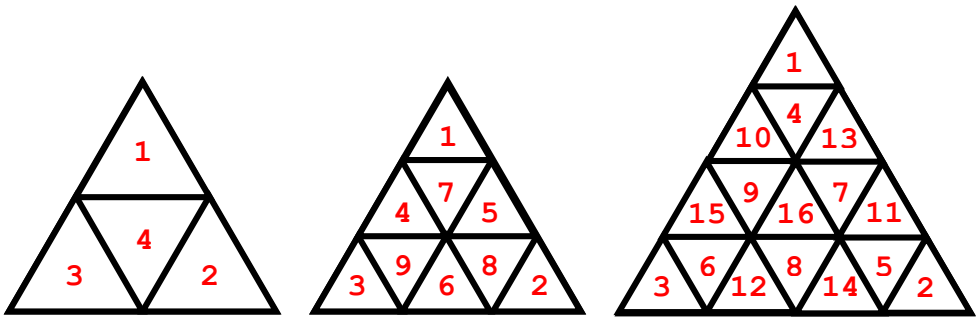
p_group_generated_by(p) - zwraca listę permutacji, które tworzą grupę generowaną przez permutację p: $[p^0, p^1, \ldots, p^{\text{rząd}(p)-1}]$

square_grid_cyclic_group(n) - zwraca grupę permutacji indukowaną w zbiorze n*n kwadratów jednostkowych kwadratu o boku n, przez grupę obrotów tego kwadratu wokół jego „przyszpilonego” środka. Kwadraty są ponumerowane kolejnymi liczbami naturalnymi, zaczynając od 1, wierszami, od lewej do prawej; np. dla n=3 mamy:

1	2	3
4	5	6
7	8	9

square_grid_dihedral_group(n) - zwraca grupę permutacji indukowaną w zbiorze n*n kwadratów jednostkowych kwadratu o boku n, przez grupę ośmiu permutacji: 4 obrotów i 4 symetrii osiowych tego kwadratu.

triangle_grid_cyclic_group(n) - zwraca grupę permutacji indukowaną w zbiorze trójkątów równobocznych o boku 1 (narysowanych w trójkącie o boku n jednostek) przez grupę trzech obrotów tego trójkąta wokół jego „przyszpilonego” środka. Trójkąty, dla n = 2, 3 i 4, są ponumerowane tak, jak to pokazano na rysunkach:



triangle_grid_dihedral_group(n) - zwraca grupę permutacji indukowaną w zbiorze trójkątów równobocznych o boku 1 (narysowanych w trójkącie o boku n jednostek) przez grupę sześciu obrotów tego trójkąta (w tym są trzy obroty „na drugą stronę"). Trójkąty, dla n = 2, 3 i 4, są ponumerowane tak, jak to pokazano na rysunkach.

---

p_group_orbits_count(grupa) - funkcja wyznacza liczbę orbit grupy permutacji

p_group_orbits(grupa) - funkcja wyznacza orbity grupy permutacji

p_group_orbit_of_x(grupa, x) - funkcja zwraca orbitę, do której należy element x, czyli zwraca te elementy zbioru X (na którym działa grupa), które są razem z elementem x w tej samej orbicie

---

p_group_cycle_index_monomial(p, zmienne) – wyznacza jednomian indeksu cyklowego dla permutacji p. Funkcja wymaga odpowiedniej listy zmiennych indeksowanych (np. [x[1], x[2], x[3], x[4]]), które wystąpią w jednomianie.

p_group_cycle_index(grupa, zmienne) - funkcja wyznacza wielomian indeksu cyklowego dla podanej grupy permutacji. Zmienne to lista zmiennych indeksowanych (np. [x[1], x[2], x[3], x[4]]) występujących w wynikowym wielomianie. Należy podać właściwą liczbę zmiennych z odpowiednimi indeksami!

p_group_cycle_index_subst(wielomian, zmienna) – funkcja podstawia w wielomianie indeksu cyklowego zmienną zmienna (zazwyczaj o nazwie k) za każdą zmienną x_i; funkcja zwraca wielomian jednej zmiennej

cyclic_group_cycle_index(n, zmienne),

dihedral_group_cycle_index(n, zmienne)

alternating_group_cycle_index(n, zmienne)

symmetric_group_cycle_index(n, zmienne) – funkcje te wyznaczają wielomiany indeksu cyklowego dla wybranych grup permutacji. Drugi argument powinien być odpowiednio liczną listą zmiennych indeksowanych, z odpowiednimi indeksami, zależnie od wybranej grupy permutacji i zależnie od n.

---

p_group_gen_fun_for_colorings_at_most(grupa, lista_kolorow) – funkcja wyznacza funkcję tworzącą ciągu liczb istotnie różnych kolorowań elementów zbioru, na którym działa grupa permutacji, za pomocą co najwyżej tylu kolorów, ile wymieniono na liście lista_kolorow. Kolory podajemy symbolami: [b, w], [r, g, b] itp.

p_group_gen_fun_for_colorings_exact(grupa, lista_kolorow) – funkcja wyznacza funkcję tworzącą ciągu liczb istotnie różnych kolorowań elementów zbioru, na którym działa grupa permutacji, za pomocą dokładnie tylu kolorów, ile wymieniono na liście lista_kolorów. Kolory podajemy symbolami: [b, w], [r, g, b] itp.

p_group_gen_fun_for_proper_colorings_at_most(grupa, graf_konfliktow, lista_kolorow) - funkcja wyznacza funkcję tworzącą ciągu liczb istotnie różnych i równocześnie prawidłowych kolorowań elementów zbioru, na którym działa grupa permutacji, za pomocą co najwyżej tylu kolorów, ile wymieniono na liście lista_kolorow. Kolorowanie prawidłowe oznacza, że żadne dwa elementy zbioru połączone krawędzią z grafu konfliktów nie będą miały tego samego koloru. Kolory podajemy symbolami: [b, w], [r, g, b] itp.

p_group_gen_fun_for_proper_colorings_exact(grupa, graf_konfliktow, lista_kolorow) - funkcja wyznacza funkcję tworzącą ciągu liczb istotnie różnych i równocześnie prawidłowych kolorowań elementów zbioru, na którym działa grupa permutacji, za pomocą dokładnie tylu kolorów, ile wymieniono na liście lista_kolorow. Kolorowanie prawidłowe oznacza, że żadne dwa elementy zbioru połączone krawędzią z grafu konfliktów nie będą miały tego samego koloru. Kolory podajemy symbolami: [b, w], [r, g, b] itp.

---

p_group_number_of_classes_of_colorings_at_most(grupa, k) – zwraca liczbę różnych klas kolorowań elementów zbioru, na którym działa grupa permutacji, za pomocą co najwyżej k kolorów. Do tej samej klasy należą wszystkie te kolorowania, w których każdy z k kolorów został

użyty konkretną liczbę razy (0 razy jest dopuszczalne). Na przykład lista [2,1,2,3] koduje klasę kolorowań za pomocą co najwyżej 4 kolorów. Lista [3,0,0,5] koduję inną klasę kolorowań za pomocą co najwyżej 4 kolorów, przy czym kolory drugi i trzeci nie zostały użyte. Łatwo obliczyć, że zbiór, na którym działa grupa permutacji, w tym przypadku, jest 8-elementowy. Liczbę klas kolorowań opisanego typu można obliczyć ze wzoru $\binom{n+k-1}{n}$, gdzie n to liczba elementów zbioru, na którym działa grupa permutacji, czyli liczba kolorowanych obiektów.

p_group_number_of_classes_of_colorings_exact(grupa, k) – zwraca liczbę różnych klas kolorowań elementów zbioru, na którym działa grupa permutacji, za pomocą dokładnie k kolorów. Do tej samej klasy należą wszystkie te kolorowania, w których każdy z k kolorów został użyty konkretną liczbę razy i każdy co najmniej raz. Na przykład lista [2,1,2,3] koduje pewną klasę kolorowań za pomocą dokładnie 4 kolorów, ale listy [4,2,0,2] oraz [3,3,2] nie kodują żadnej klasy kolorowań za pomocą dokładnie 4 kolorów. Łatwo obliczyć, że zbiór, na którym działa grupa permutacji, jest 8-elementowy. Liczbę klas kolorowań opisanego typu można obliczyć ze wzoru $\binom{n-1}{k-1}$, gdzie n to liczba elementów zbioru, na którym działa grupa permutacji, czyli liczba kolorowanych obiektów.

---

p_group_number_of_colorings_at_most(grupa, k) - zwraca wielomian zmiennej k (albo konkretną wartość, gdy k jest liczbą), który określa liczbę istotnie różnych kolorowań elementów zbioru, na którym działa grupa permutacji, za pomocą co najwyżej k kolorów.

p_group_number_of_colorings_exact(grupa, k) - zwraca liczbę istotnie różnych kolorowań elementów zbioru, na którym działa grupa permutacji, za pomocą dokładnie k kolorów. Użycie zmiennej k w postaci symbolu daje sumę, która jest mało przydatna.

p_group_number_of_colorings_exact2(grupa, kolory) – zwraca liczbę istotnie różnych kolorowań elementów zbioru, na którym działa grupa permutacji, za pomocą podanej listy użyć każdego koloru. Funkcja działa w ten sposób, że wyznacza współczynnik liczbowy stojący przy jednomianie, zakodowanym listą o nazwie kolory, w funkcji tworzącej ciągu liczb istotnie różnych kolorowań, powstałej z indeksu cyklowego tej grupy, poprzez podstawienie za każdą zmienną x_i sumy $r^i + g^i + b^i + \ldots$ Zmienna kolory (np. [2,4,1]) określa, o który jednomian chodzi. W tym przykładzie chodzi o współczynnik stojący przy jednomianie $r^2g^4b^1$ (gdyby kolory nazwano r, g, b).

---

p_group_number_of_proper_colorings_at_most(grupa, graf_konfliktow, k, [info]) – zwraca wielomian zmiennej k (albo konkretną wartość, gdy k jest liczbą), z którego można obliczyć liczbę istotnie różnych kolorowań elementów zbioru, na którym działa grupa, za pomocą co najwyżej k kolorów, przy założeniu, że kolorowania będą prawidłowe, tzn. pary elementów zbioru (na którym działa grupa), które stanowią krawędź w grafie konfliktów, nie będą miały tego samego koloru. Użycie opcjonalnego argumentu info pozwala zobaczyć cząstkowe wyniki działania tej funkcji, tzn. wielomiany dla każdej permutacji.

p_group_number_of_proper_colorings_exact(grupa, graf_konfliktow, k) - zwraca liczbę istotnie różnych kolorowań elementów zbioru, na którym działa grupa, za pomocą dokładnie k kolorów, przy założeniu, że kolorowania będą prawidłowe, tzn. pary elementów zbioru, które stanowią krawędź w grafie konfliktów, nie będą miały tego samego koloru.

p_group_number_of_proper_colorings_exact2(grupa, graf_konfliktow, kolory) - zwraca liczbę istotnie różnych prawidłowych kolorowań elementów zbioru, na którym działa grupa, za pomocą podanej listy liczb użyć poszczególnych kolorów. Pary elementów zbioru, które stanowią krawędź w grafie konfliktów, nie będą miały tego samego koloru.

p_group_number_of_proper_colorings_at_most2(grupa, graf_konfliktow, zmienne, [info]) – zwraca wielomian wielu zmiennych indeksowanych. Jeżeli za każdą zmienną indeksowaną podstawić np. k, to otrzymamy wielomian taki sam, jak z funkcji bez dopisanej liczby 2. Najważniejsze zastosowanie tego wielomianu wielu zmiennych dotyczy wyznaczenia funkcji tworzącej ciągu liczb kolorowań istotnie różnych i równocześnie prawidłowych, zgodnie z grafem konfliktów. Opcjonalny parametr info pozwala zobaczyć wielomian dla każdej permutacji.

---

p_group_all_colorings_at_most(grupa, k) - zwraca listę wszystkich istotnie różnych kolorowań elementów zbioru, na którym działa grupa, za pomocą co najwyżej k kolorów.

p_group_all_colorings_exact(grupa, k) - zwraca listę wszystkich istotnie różnych kolorowań elementów zbioru, na którym działa grupa, za pomocą dokładnie k kolorów.

p_group_all_colorings_exact2(grupa, kolory) - zwraca listę wszystkich istotnie różnych kolorowań elementów zbioru, na którym działa grupa, za pomocą tylu użyć poszczególnych kolorów, ile wymieniono na liście kolory (np. [2,3,2,1]). Suma liczb na liście kolory musi być równa liczbie elementów zbioru, na którym działa grupa. Ilość liczb na liście kolory określa liczbę różnych kolorów.

---

p_group_all_proper_colorings_at_most(grupa, graf_konfliktow, k) - zwraca listę wszystkich istotnie różnych prawidłowych kolorowań elementów zbioru, na którym działa grupa, za pomocą co najwyżej k kolorów. Graf konfliktów określa, które pary elementów zbioru sąsiadują ze sobą (umownie). Żadne dwa sąsiadujące elementy nie mogą być pokolorowane tym samym kolorem.

p_group_all_proper_colorings_exact(grupa, graf_konfliktow, k) – zwraca listę wszystkich istotnie różnych prawidłowych kolorowań elementów zbioru, na którym działa grupa, za pomocą dokładnie k kolorów. Graf konfliktów określa, które pary elementów zbioru sąsiadują ze

sobą (umownie). Żadne dwa sąsiadujące elementy nie mogą być pokolorowane tym samym kolorem.

p_group_all_proper_colorings_exact2(grupa, graf_konfliktow, kolory) – zwraca listę wszystkich istotnie różnych prawidłowych kolorowań elementów zbioru, na którym działa grupa, za pomocą tylu kolorów, ile wymieniono na liście kolory (na tej liście podaje się liczby wystąpień poszczególnych kolorów).Graf konfliktów określa, które elementy zbioru sąsiadują ze sobą (umownie). Funkcja ta wyznacza kolorowania prawidłowe, tzn. że żadne dwa sąsiadujące elementy zbioru nie będą pokolorowane tym samym kolorem.

---

graph_all_unique_proper_vertices_colorings_at_most(graf, k) – zwraca listę wszystkich istotnie różnych (ze względu na grupę automorfizmów) prawidłowych kolorowań wierzchołków grafu graf za pomocą co najwyżej k kolorów. Graf podajemy jako listę krawędzi. Każda krawędź to lista 2-elementowa. Graf musi być nieskierowany, bez pętli i bez krawędzi wielokrotnych oraz bez izolowanych wierzchołków. Graf może być niespójny.

graph_all_unique_proper_vertices_colorings_exact(graf, k) – zwraca listę wszystkich istotnie różnych (ze względu na grupę automorfizmów) prawidłowych kolorowań wierzchołków grafu graf za pomocą dokładnie k kolorów.

graph_all_unique_proper_vertices_colorings_exact2(graf, kolory) - zwraca listę wszystkich istotnie różnych (ze względu na grupę automorfizmów) prawidłowych kolorowań wierzchołków grafu graf za pomocą tylu kolorów, ile wymieniono na liście kolory. Na tej liście podaje się liczby wystąpień poszczególnych kolorów (np. [2,1,3,2] oznacza, że użyto 4 kolory).

---

graph_all_unique_proper_edges_colorings_at_most(graf, k) – zwraca listę wszystkich istotnie różnych (ze względu na grupę permutacji indukowaną w zbiorze krawędzi przez grupę automorfizmów) prawidłowych kolorowań krawędzi grafu graf za pomocą co najwyżej k kolorów.

graph_all_unique_proper_edges_colorings_exact(graf, k) – zwraca listę wszystkich istotnie różnych (ze względu na grupę permutacji indukowaną w zbiorze krawędzi przez grupę automorfizmów) prawidłowych kolorowań krawędzi grafu graf za pomocą dokładnie k kolorów.

graph_all_unique_proper_edges_colorings_exact2(graf, kolory) – zwraca listę wszystkich istotnie różnych (ze względu na grupę permutacji indukowaną w zbiorze krawędzi przez grupę automorfizmów) prawidłowych kolorowań krawędzi grafu graf za pomocą tylu kolorów, ile wymieniono na liście kolory. Na tej liście podaje się liczby wystąpień kolorów (np. [2,1,3,2]).

---

graph_automorphism_group(graf) – zwraca grupę permutacji, które są automorfizmami grafu graf; jako kryterium podziału wierzchołków na rozłączne podzbiory przyjęto stopień wierzchołka

graph_automorphism_group2(graf) – zwraca grupę permutacji, które są automorfizmami grafu graf; jako kryterium podziału wierzchołków na rozłączne podzbiory przyjęto sekwencję stopni wierzchołków sąsiadów. Ta funkcja powinna być trochę szybsza niż poprzednia.

digraph_automorphism_group(digraf) – zwraca grupę permutacji, które są automorfizmami grafu skierowanego digraf; jako kryterium podziału wierzchołków na rozłączne podzbiory przyjęto parę liczb: stopień wychodzący i stopień wchodzący wierzchołka.

---

graph_edges_induced_group(graf) – zwraca grupę permutacji indukowaną w zbiorze krawędzi przez grupę automorfizmów grafu graf; krawędzie są oznaczone kolejnymi liczbami naturalnymi, zgodnie z kolejnością ich występowania na liście graf

graph_all_edges_induced_group(graf) – zwraca grupę permutacji indukowaną w zbiorze wszystkich krawędzi (par wierzchołków) przez grupę automorfizmów grafu graf; krawędzie grafu są oznaczone kolejnymi liczbami natural-

nymi, zgodnie z kolejnością ich występowania na liście graf. Pozostałe krawędzie mają kolejne numery. Funkcja zwraca listę 2-elementową. Pierwszym elementem jest wspomniana grupa permutacji, drugim – lista wszystkich krawędzi.

digraph_arcs_induced_group(digraf) – zwraca grupę permutacji indukowaną w zbiorze łuków przez grupę automorfizmów grafu skierowanego digraf; łuki są oznaczone kolejnymi liczbami naturalnymi, zgodnie z kolejnością ich występowania na liście digraf

digraph_all_arcs_induced_group(digraf) – zwraca grupę permutacji indukowaną w zbiorze wszystkich łuków (uporządkowanych par wierzchołków) przez grupę automorfizmów grafu skierowanego digraf; łuki digrafu są oznaczone kolejnymi liczbami naturalnymi, zgodnie z kolejnością ich występowania na liście digraf. Pozostałe łuki mają kolejne numery. Funkcja zwraca listę 2-elementową. Pierwszym elementem jest wspomniana grupa permutacji, drugim – lista wszystkich łuków.

---

graph_automorphism_group_chromatic_polynomial(graf, zmienna, [info]) – zwraca wielomian zmiennej zmienna, określający liczbę istotnie różnych (ze względu na grupę automorfizmów grafu) prawidłowych kolorowań wierzchołków grafu za pomocą co najwyżej k kolorów (jeżeli jako zmienną podamy k, a nie konkretną liczbę). Użycie opcjonalnego argumentu info pozwala zobaczyć cząstkowe wyniki działania tej funkcji, tzn. wielomiany dla każdej permutacji.

graph_edges_induced_group_chromatic_polynomial(graf, zmienna, [info]) – zwraca wielomian zmiennej zmienna, określający liczbę istotnie różnych (ze względu na grupę permutacji indukowaną w zbiorze krawędzi przez grupę automorfizmów grafu) prawidłowych kolorowań krawędzi grafu za pomocą co najwyżej k kolorów (jeżeli w miejsce zmiennej podamy zmienną k). Użycie opcjonalnego argumentu info pozwala zobaczyć cząstkowe wyniki działania tej funkcji,

tzn. wielomiany dla każdej permutacji indukowanej w zbiorze krawędzi.

---

all_unique_rooks_placements(n) – wyznacza wszystkie rozmieszczenia n wzajemnie nieatakujących się wież na szachownicy n x n, które są unikalne ze względu na grupę czterech obrotów i czterech symetrii osiowych szachownicy (OEIS A000903). Każde rozstawienie wież jest zakodowane permutacją numerów kolumn, w których znajdują się wieże. Dla n=8 obliczenia trwają około 5 minut.

p_group_all_unique_rooks_placements(n) – zwraca grupę złożoną z ośmiu permutacji, indukowaną w zbiorze n! rozstawień n wzajemnie nieatakujących się wież na szachownicy n x n, przez grupę ośmiu obrotów tej szachownicy (brak obrotu, obroty o 90, 180 i 270 stopni oraz 4 symetrie osiowe: \, /, |, --). Każde rozstawienie n wież jest zakodowane liczbą naturalną o 1 większą od numeru (w porządku leksykograficznym) permutacji utworzonej z numerów kolumn (w zapisie 1-wierszowym), w których znajdują się wieże. Aby dla numeru nr zobaczyć rozstawienie wież, należy (dla konkretnego n) wyznaczyć postać macierzową permutacji o numerze nr-1 w Sn: perm2matrix(p_from_number(nr-1,n)).

all_unique_rooks_placements2(n) – wyznacza wszystkie rozmieszczenia n wzajemnie nieatakujących się wież na szachownicy n x n, które są unikalne ze względu na grupę czterech obrotów szachownicy (OEIS A263685): brak obrotu, obroty o 90, 180 i 270 stopni. Każde rozstawienie wież jest zakodowane permutacją numerów kolumn, w których znajdują się wieże. Dla n=8 obliczenia trwają około 4 minut.

p_group_all_unique_rooks_placements2(n) – zwraca grupę złożoną z czterech permutacji, indukowaną w zbiorze n! rozstawień n wzajemnie nieatakujących się wież na szachownicy n x n, przez grupę czterech obrotów tej szachownicy (brak obrotu oraz obroty o 90, 180 i 270 stopni). Każde rozstawienie n wież jest zakodowa-

ne liczbą naturalną o 1 większą od numeru (w porządku leksykograficznym) permutacji utworzonej z numerów kolumn (w zapisie 1-wierszowym), w których znajdują się wieże. Aby dla numeru nr zobaczyć rozstawienie wież, należy (dla konkretnego n) obliczyć: perm2matrix(p_from_number(nr-1,n)).

---

p_group_hexaedr_vertices()

p_group_hexaedr_edges()

p_group_hexaedr_faces() - trzy grupy permutacji indukowane w zbiorach wierzchołków, krawędzi i ścian sześcianu przez grupę 24 obrotów tego sześcianu

p_group_tetraedr_vertices()

p_group_tetraedr_edges()

p_group_tetraedr_faces() - trzy grupy permutacji indukowane w zbiorach wierzchołków, krawędzi i ścian czworościanu przez grupę 12 obrotów tego czworościanu

p_group_two_tetraedrs_vertices()

p_group_two_tetraedrs_edges()

p_group_two_tetraedrs_faces() - trzy grupy permutacji indukowane w zbiorach wierzchołków, krawędzi i ścian bryły powstałej ze sklejenia dwóch czworościanów foremnych, przez grupę 6 obrotów tej bryły

p_group_right_cuboid_vertices()

p_group_right_cuboid_edges()

p_group_right_cuboid_faces() - trzy grupy permutacji indukowane przez grupę 8 obrotów w zbiorach wierzchołków, krawędzi i ścian prostopadłościanu o podstawie kwadratowej (pozostałe 4 ściany nie są kwadratami!)

p_group_not_right_cuboid_vertices()

p_group_not_right_cuboid_edges()

p_group_not_right_cuboid_faces() - trzy grupy permutacji indukowane przez grupę 4 obrotów w zbiorach wierzchołków, krawędzi i ścian prostopadłościanu, w którym żadna ściana nie jest kwadratem

p_group_isometries_hexaedr_vertices()

p_group_isometries_hexaedr_edges()

p_group_isometries_hexaedr_faces() – trzy grupy permutacji indukowane przez grupę 48 izometrii sześcianu w zbiorach wierzchołków, krawędzi i ścian sześcianu

p_group_triangular_cuboid_vertices()

p_group_triangular_cuboid_edges()

p_group_triangular_cuboid_faces() – trzy grupy permutacji indukowane przez grupę 6 obrotów w zbiorach wierzchołków, krawędzi i ścian prostopadłościanu, w którym obie podstawy są trójkątami równobocznymi

---

line_graph(graf) – wyznacza graf krawędziowy dla nieskierowanego grafu prostego (graf powinien być spójny)

complete_graph(n) – zwraca graf pełny K_n

complete_digraph(n) – zwraca digraf, który powstanie z K_n, gdy każdą jego krawędź zastąpimy parą przeciwnych łuków

cycle_graph(n) – zwraca graf cykliczny C_n

prism_graph(n) – zwraca graf pryzmowy Y_n

path_graph(n) – zwraca graf ścieżkowy P_n

star_graph(n) – zwraca graf gwiazdowy S_n

book_graph(n) – zwraca graf książkowy B_n

ladder_graph(n) – zwraca graf drabinkowy L_n

wheel_graph(n) – zwraca graf kołowy W_n

petersen_graph() – zwraca graf Petersena

hexaedr_graph() – zwraca graf utworzony z wierzchołków i krawędzi sześcianu

hexaedr_faces_adjacency_graph() – zwraca graf sąsiedztw ścian sześcianu

is_edge(graf, krawędź) – zwraca true, jeżeli w grafie jest krawędź

is_arc(digraf, łuk) – zwraca true, jeżeli w digrafie jest łuk

contract_edge(graf, kraw) – zwraca graf, w którym dwa wierzchołki wymienione na liście kraw zostały połączone ze sobą w je-

den wierzchołek; jeżeli krawędź kraw istniała w grafie, to zostanie usunięta

rook_graph(n) - dla kwadratowej siatki n x n, w której pola są ponumerowane wierszami od lewej do prawej, zaczynając od 1, zwraca graf, w którym wierzchołkami są pola tej siatki, a krawędź istnieje między dwoma wierzchołkami, jeżeli oba pola leżą w tym samym wierszu lub w tej samej kolumnie

square_grid_adjacency_graph(n) - dla kwadratowej siatki n x n, w której pola są ponumerowane wierszami od lewej do prawej, zaczynając od 1, zwraca graf, w którym wierzchołkami są pola siatki, a krawędź istnieje między dwoma wierzchołkami, jeżeli oba pola sąsiadują ze sobą, czyli mają wspólny bok (krawędź) o długości jeden

---

all_nonisomorphic_undirected_graphs(n,[m]) – jeżeli drugi, opcjonalny argument jest pominięty, wtedy funkcja zwraca listę wszystkich parami nieizomorficznych nieskierowanych nieoznakowanych grafów prostych n-wierzchołkowych (OEIS A000088). Natomiast, jeżeli argument m jest użyty, wtedy funkcja zwraca tylko m-krawędziowe grafy opisanego typu. Funkcja działa w ten sposób, że najpierw wyznacza grupę n! permutacji indukowaną w zbiorze krawędzi grafu pełnego K_n przez grupę automorfizmów tego grafu (przez grupę symetryczną S_n), a następnie wyznacza istotnie różne kolorowania krawędzi tego grafu za pomocą co najwyżej dwóch kolorów: czarnego i białego. Krawędzie pokolorowane na biało są pomijane, a pokolorowane na czarno zostają w grafie. Każde kolorowanie określa jeden unikalny graf nieskierowany. Jeżeli np. interesują nas tylko 3-krawędziowe grafy o 4 wierzchołkach, to można je otrzymać generując wszystkie grafy i odfiltrowując 3-krawędziowe, ale to trwa dłużej niż użycie tej funkcji z argumentami (4,3).

```
(%i8)   grafy : all_nonisomorphic_undirected_graphs(4)$
        sublist(grafy, lambda([g],length(g)=3));
(%o8)   [[[1,2],[1,3],[1,4]],[[1,2],[1,3],[2,3]],[[1,2],[1,3],[2,4]]]

(%i9)   all_nonisomorphic_undirected_graphs(4,3);
(%o9)   [[[1,2],[1,3],[1,4]],[[1,2],[1,3],[2,3]],[[1,2],[1,3],[2,4]]]
```

all_nonisomorphic_directed_graphs(n,[m]) - jeżeli drugi, opcjonalny argument jest pominięty, wtedy funkcja zwraca listę wszystkich parami nieizomorficznych skierowanych nieoznakowanych grafów prostych n-wierzchołkowych (OEIS A000273). Funkcja działa w ten sposób, że najpierw wyznacza grupę n! permutacji indukowaną w zbiorze łuków przez grupę automorfizmów digrafu pełnego K_n (przez grupę symetryczną S_n), a następnie wyznacza istotnie różne kolorowania łuków tego grafu za pomocą co najwyżej dwóch kolorów: czarnego i białego. Łuki pokolorowane na biało są pomijane, a pokolorowane na czarno zostają w digrafie. Każde kolorowanie określa jeden unikalny digraf. Jeżeli np. interesują nas tylko 2-łukowe digrafy o 4 wierzchołkach, to można je odfiltrować po wygenerowaniu wszystkich digrafów, ale trwa to znacznie dłużej niż użycie opisywanej funkcji z argumentami (4,2).

```
(%i11)  digrafy : all_nonisomorphic_directed_graphs(4)$
        sublist(digrafy, lambda([g],length(g)=2));
(%o11)  [[[1,2],[1,3]],[[1,2],[2,1]],[[1,2],[2,3]],[[1,2],[3,2]],[[1,2],[3,4]]]

(%i12)  all_nonisomorphic_directed_graphs(4,2);
(%o12)  [[[1,2],[1,3]],[[1,2],[2,1]],[[1,2],[2,3]],[[1,2],[3,2]],[[1,2],[3,4]]]
```

---

number_of_graph_vertices_labelings(graf) - funkcja zwraca liczbę istotnie różnych (ze względu na grupę automorfizmów grafu) sposobów oznakowania n wierzchołków grafu nieskierowanego liczbami od 1 do n.

number_of_graph_edges_labelings(graf) - funkcja zwraca liczbę istotnie różnych (ze względu na grupę permutacji indukowaną w zbiorze krawędzi przez grupę automorfizmów grafu) sposobów oznakowania m krawędzi grafu nieskierowanego liczbami od 1 do m.

number_of_digraph_vertices_labelings(digraf) - funkcja zwraca liczbę istotnie różnych (ze względu na grupę automorfizmów

digrafu) sposobów oznakowania n wierzchołków grafu skierowanego liczbami od 1 do n.

number_of_digraph_arcs_labelings(digraf) – funkcja zwraca liczbę istotnie różnych (ze względu na grupę permutacji indukowaną w zbiorze łuków przez grupę automorfizmów grafu) sposobów oznakowania m łuków grafu skierowanego liczbami od 1 do m.

---

number_of_free_necklaces(kolory) - funkcja zwraca liczbę różnych tzw. wolnych naszyjników, jakie można utworzyć z kolorowych koralików. kolory to lista liczb koralików w tym samym kolorze. Na przykład [2,2,1,3] oznacza, że naszyjnik będzie posiadał 8 koralików w czterech kolorach. Wolne naszyjniki można podnosić ze stołu i odwracać oraz, oczywiście, można dowolnie przesuwać koraliki po sznurku.

all_free_necklaces(kolory) - funkcja zwraca listę wszystkich wolnych naszyjników, jakie można utworzyć z kolorowych koralików. Każdy naszyjnik to lista kolorów poszczególnych koralików. Argument kolory to lista liczb koralików w tym samym kolorze. Na przykład [2,2,1,3] oznacza, że naszyjnik będzie posiadał 8 koralików w czterech kolorach. Wolne naszyjniki można podnosić ze stołu i odwracać oraz, oczywiście, można przesuwać w nich koraliki po sznurku.

number_of_fixed_necklaces(kolory) - funkcja zwraca liczbę różnych tzw. zafiksowanych naszyjników, jakie można utworzyć z kolorowych koralików. kolory to lista liczb koralików w tym samym kolorze. Na przykład [2,2,1,3] oznacza, że naszyjnik będzie posiadał 8 koralików w czterech kolorach. Zafiksowanych naszyjników nie można podnosić ze stołu i odwracać, ale można w nich przesuwać koraliki po sznurku.

all_fixed_necklaces(kolory) - funkcja zwraca listę wszystkich zafiksowanych naszyjników, jakie można utworzyć z kolorowych koralików. Każdy obliczony naszyjnik jest zakodowany w postaci listy kolorów poszczególnych kora-

lików. Argument kolory musi być listą liczb koralików w tym samym kolorze. Na przykład [2,2,1,3] oznacza, że naszyjnik będzie posiadał 8 koralików w czterech kolorach. Zafiksowanych naszyjników nie można podnosić ze stołu i odwracać, ale można w nich przesuwać koraliki po sznurku.

------Poniższe funkcje nie mają nic wspólnego z teorią grup permutacji---------

number_of_proper_vertices_colorings_at_most(graf, k) – zwraca liczbę prawidłowych kolorowań wierzchołków grafu za pomocą co najwyżej k kolorów; k może być zmienną lub liczbą. Funkcja korzysta ze zwyczajnego wielomianu chromatycznego.

number_of_proper_vertices_colorings_exact(graf, k) – zwraca liczbę prawidłowych kolorowań wierzchołków grafu za pomocą dokładnie k kolorów; k powinno być liczbą. Funkcja korzysta ze zwyczajnego wielomianu chromatycznego oraz ze wzoru na Cdok(k).

number_of_proper_vertices_colorings_exact2(graf, kolory) – zwraca liczbę prawidłowych kolorowań wierzchołków grafu za pomocą zadanego zestawu kolorów. W liście kolory (np. [3, 2, 2]) podaje się liczby użyć poszczególnych kolorów. Długość listy kolory jest równa liczbie różnych kolorów, a suma liczb musi być równa liczbie wierzchołków grafu (które kolorujemy). Funkcja działa w oparciu o wielomian chromatyczny wielu zmiennych.

number_of_proper_edges_colorings_at_most(graf, k) – zwraca liczbę prawidłowych kolorowań krawędzi grafu graf za pomocą co najwyżej k kolorów; k może być zmienną lub liczbą.

number_of_proper_edges_colorings_exact(graf, k) - zwraca liczbę prawidłowych kolorowań krawędzi grafu graf za pomocą dokładnie k kolorów; k powinno być liczbą.

number_of_proper_edges_colorings_exact2(graf, kolory) - zwraca liczbę prawidłowych kolorowań krawędzi grafu za pomocą zadanego zestawu kolorów. W liście kolory (np. [3, 2, 2]) podaje się liczby użyć poszczególnych kolorów. Długość listy kolory jest równa liczbie różnych kolorów, a suma liczb musi być równa liczbie krawędzi grafu (które kolorujemy).

------Poniższe funkcje nie mają nic wspólnego z teorią grup permutacji---------

chromatic_poly(graf, k) – wyznacza zwyczajny wielomian chromatyczny grafu, z którego można obliczyć liczbę prawidłowych kolorowań zaetykietowanych wierzchołków grafu za pomocą co najwyżej k kolorów. Jeżeli k jest liczbą, to funkcja zwraca wartość wielomianu.

chromatic_poly2(graf, zmienne) - wyznacza wielomian chromatyczny nie jednej, ale wielu zmiennych. Jeżeli za każdą zmienną indeksowaną w tym wielomianie podstawimy zmienną k, to otrzymamy zwyczajny wielomian chromatyczny. Można także za każdą zmienną indeksowaną podstawić (taką samą) konkretną liczbę, wówczas otrzymamy liczbę prawidłowych kolorowań wierzchołków grafu za pomocą co najwyżej podanej liczby kolorów. Najważniejszym zastosowaniem tego wielomianu jest to, że jeżeli za każdą zmienną indeksowaną k_i podstawimy sumę $r^i + g^i + b^i + \ldots$, wtedy dostaniemy funkcję tworzącą ciągu liczb prawidłowych kolorowań wierzchołków grafu za pomocą co najwyżej tylu kolorów, ile zmiennych r, g, b,… użyto w podstawieniu. Argument zmienne musi być listą zmiennych indeksowanych (np. [k[1], k[2], k[3]], o indeksach od 1 do liczby wierzchołków grafu. Aby wyznaczyć ten wielomian chromatyczny wielu zmiennych, należy zastosować na grafie wielokrotnie procedurę usuwania i zwierania krawędzi, i dojść do grafów pustych zachowując multiwierzchołki.

chromatic_poly2_subst(chrompoly, zmienna) – w wielomianie zwróconym przez funkcję chromatic_poly2(graf, zmienne) podstawia zmienną (np. k) za każdą zmienną indeksowaną. Można też podstawić konkretną liczbę.

------Poniższe funkcje nie mają nic wspólnego z teorią grup permutacji---------

graph_gen_fun_for_proper_vertices_colorings_at_most(graf, lista_kolorow) – zwraca funkcję tworzącą ciągu liczb kolorowań wierzchołków grafu za pomocą co najwyżej tylu kolorów, ile zmiennych podano na liście lista_ kolorow (np. [r, g, b]). Funkcja tworząca to wielomian wielu zmiennych, w którym zmiennymi

niezależnymi są te nazwy kolorów. Każdy składnik tego wielomianu koduje jedną rodzinę kolorowań, do której należą wszystkie te kolorowania, w których użyto takiego samego zestawu kolorów. Wykładniki zmiennych kodują liczby użyć poszczególnych kolorów w rodzinie, a współczynnik danego składnika (jednomianu) to liczba kolorowań w tej rodzinie.

graph_gen_fun_for_proper_vertices_colorings_exact(graf, lista_kolorow) - zwraca funkcję tworzącą ciągu liczb kolorowań wierzchołków grafu za pomocą dokładnie tylu kolorów, ile zmiennych podano na liście lista_kolorow (np. [r, g, b]). Określenie *dokładnie* oznacza, że każdy kolor musi być użyty co najmniej raz (co najmniej jeden wierzchołek musi być w każdym kolorze).

graph_gen_fun_for_proper_edges_colorings_at_most(graf, lista_kolorow) – zwraca funkcję tworzącą ciągu liczb kolorowań krawędzi grafu za pomocą co najwyżej tylu kolorów, ile zmiennych zawiera lista_ kolorow (np. [r, g, b]).

graph_gen_fun_for_proper_edges_colorings_exact(graf, lista_kolorow) - zwraca funkcję tworzącą ciągu liczb kolorowań krawędzi grafu za pomocą dokładnie tylu kolorów, ile zmiennych zawiera lista_ kolorow (np. [r, g, b]).

------Poniższe funkcje nie mają nic wspólnego z teorią grup permutacji---------

graph_all_proper_vertices_colorings_at_most(graf, k) - zwraca listę wszystkich prawidłowych kolorowań wierzchołków grafu graf za pomocą co najwyżej k kolorów.

graph_all_proper_vertices_colorings_exact(graf, k) – zwraca listę wszystkich prawidłowych kolorowań wierzchołków grafu graf za pomocą dokładnie k kolorów.

graph_all_proper_vertices_colorings_exact2(graf, kolory) – zwraca listę wszystkich prawidłowych kolorowań wierzchołków grafu graf za pomocą tylu kolorów, ile wymieniono liczb na liście kolory. Liczby na liście kolory oznaczają liczby użyć poszczególnych kolorów (np. [2,1,3,1] oznacza, że użyto cztery kolory).

graph_all_proper_edges_colorings_at_most(graf, k) - zwraca listę wszystkich prawidłowych kolorowań krawędzi grafu graf za pomocą co najwyżej k kolorów.

graph_all_proper_edges_colorings_exact(graf, k) - zwraca listę wszystkich prawidłowych kolorowań krawędzi grafu graf za pomocą dokładnie k kolorów.

graph_all_proper_edges_colorings_exact2(graf, kolory) - zwraca listę wszystkich prawidłowych kolorowań krawędzi grafu graf za pomocą tylu kolorów, ile wymieniono liczb na liście kolory. Każda liczba na liście kolory oznacza liczbę użyć konkretnego koloru (np. [2,1,3,1] oznacza użycie czterech kolorów).

- operator składania permutacji

@ - operator potęgowania permutacji; w przypadku wykładnika w postaci ułamka zwykłego, należy ten ułamek zapisać w nawiasach, np. p@(-2/3)

|! !| - operator wyznaczania liczby permutacji danego typu

== - operator porównania dwóch permutacji (zwraca true, jeżeli dwie permutacje są równe)

---

all_solutions(rownanie, k, warunki) - zwraca liczbę wszystkich rozwiązań (w liczbach całkowitych nieujemnych) algebraicznego równania liniowego jednej lub wielu zmiennych. Współczynniki (liczby naturalne) stojące przy niewiadomych podaje się w liście rownanie, a wartość prawej strony równania - w zmiennej k. Na zmienne występujące w równaniu można nałożyć dowolny warunek logiczny. Przykład: wyznaczyć liczbę rozwiązań równania 2*x + y + z = 14, jeżeli muszą być spełnione warunki: $x \geq 2$ i $y < 7$ i $2 \leq z \leq 6$. Rozwiązanie ma postać:

```
(%i105) warunki(x) := (x[1]>=2) and (x[2]<7) and (x[3]>=2) and (x[3]<=6);
        wynik : all_solutions([2,1,1], 14, warunki);
(%o104) warunki(x) := x[1] >= 2 and x[2] < 7 and x[3] >= 2 and x[3] <= 6
(wynik) 17
```

Równanie posiada więc 17 rozwiązań. Zmienna x,

występująca jako argument funkcji warunki(x), ma formę listy. Pierwsza zmienna na tej liście, czyli x[1], odpowiada pierwszej zmiennej występującej w równaniu (tu jest to x). Druga zmienna, czyli x[2], odpowiada drugiej zmiennej występującej w równaniu (tu jest to y), itd. Lista [2,1,1] zawiera współczynniki liczbowe stojące przy zmiennych po lewej stronie równania **2***x + **1***y + **1***z = 14. Jeżeli wywołamy funkcję all_solutions() z czterema argumentami (czwarty dowolny), to jako wynik zobaczymy wszystkie rozwiązania danego równania. W tym przypadku:

```
(%i107)  warunki(x) := (x[1]>=2) and (x[2]<7) and (x[3]>=2) and (x[3]<=6);
         wynik : all_solutions([2,1,1], 14, warunki, pokaz);
(%o106)  warunki(x) := x1>=2 and x2<7 and x3>=2 and x3<=6
(wynik)  [[6,0,2],[5,2,2],[5,1,3],[5,0,4],[4,4,2],[4,3,3],[4,2,4],[4,1,5],[
4,0,6],[3,6,2],[3,5,3],[3,4,4],[3,3,5],[3,2,6],[2,6,4],[2,5,5],[2,4,6]]
```

Każda lista 3-elementowa powyżej to jedno rozwiązanie (x, y, z) danego równania z zadanymi warunkami. W funkcji warunki(x) można użyć operatorów and, or, xor, not, a także innych funkcji zwracających wartość typu logicznego (np. evenp(x[2]), oddp(x[2])). Kolejność działań logicznych w definicji funkcji warunki(x) można wymusić nawiasami okrągłymi. Priorytet operatorów logicznych, w kolejności od najwyższego, to: not, and, xor, or. Przykład równania z bardziej złożonym warunkiem logicznym:

```
(%i143)  warunki(x) := (x[1]>=6) and (x[2]<5) or ((x[3]>=1) xor evenp(x[3]));
         wynik : all_solutions([2,1,1], 14, warunki);
(%o142)  warunki(x) := x1>=6 and x2<5 or x3>=1 xor evenp(x3)
(wynik)  37
(%i145)  warunki(x) := (x[1]>=6) and (x[2]<5) or ((x[3]>=1) xor evenp(x[3]));
         wynik : all_solutions([2,1,1], 14, warunki, pokaz);
(%o144)  warunki(x) := x1>=6 and x2<5 or x3>=1 xor evenp(x3)
(wynik)  [[7,0,0],[6,2,0],[6,1,1],[6,0,2],[5,4,0],[5,3,1],[5,1,3],[4,6,0],[4,5,
1],[4,3,3],[4,1,5],[3,8,0],[3,7,1],[3,5,3],[3,3,5],[3,1,7],[2,10,0],[2,9,1],[2,
7,3],[2,5,5],[2,3,7],[2,1,9],[1,12,0],[1,11,1],[1,9,3],[1,7,5],[1,5,7],[1,3,9],
[1,1,11],[0,14,0],[0,13,1],[0,11,3],[0,9,5],[0,7,7],[0,5,9],[0,3,11],[0,1,13]]
```

Analityczne wyznaczenie liczby rozwiązań podanego równania wymaga zastosowania metody zwyczajnej funkcji tworzącej. Jeżeli warunek logiczny, nałożony na niewiadome, nie jest prostą koniunkcją warunków, wtedy rozwiązanie

analityczne wymaga wyprowadzenia kilku funkcji tworzących, a następnie skorzystania z twierdzeń na liczebność różnicy, sumy, różnicy symetrycznej zbiorów itp.

counting_words(k, n, warunki) - zwraca liczbę wszystkich wyrazów k-literowych, jakie można utworzyć z n liter. Każdy wyraz spełnia warunki nałożone na liczbę wystąpień każdej z n liter. Każdy wyraz jest więc wariacją k-elementową z powtórzeniami zbioru n-elementowego z warunkami nałożonymi na liczbę wystąpień każdej z n liter. Funkcja warunki(w) określa warunek logiczny, jaki muszą spełnić liczby wystąpień każdej z n liter. w[1], w[2], w[3] itd. to liczby wystąpień kolejnych liter. Przykład użycia funkcji zliczania wyrazów:

```
(%i170) warunki(w) := (w[1]>=2) and (w[2]=2) and (w[3]<=2);
        liczba_wyrazow : counting_words(5,3,warunki);
(%o169) warunki(w) := w1 >= 2 and w2 = 2 and w3 <= 2
(liczba_wyrazow) 40
```

Jeżeli wywołać funkcję counting_words() z czterema argumentami (czwarty dowolny), wtedy zwróci ona nie liczbę, ale listę wszystkich wyrazów spełniających zadane warunki logiczne:

```
(%i176) warunki(w) := (w[1]>=2) and (w[2]=2) and (w[3]<=2);
        liczba_wyrazow : counting_words(5,3,warunki,pokaz);
(%o175) warunki(w) := w1 >= 2 and w2 = 2 and w3 <= 2
(liczba_wyrazow) [[1,1,1,2,2],[1,1,2,1,2],[1,1,2,2,1],[1,1,2,2,3],[1,1,
2,3,2],[1,1,3,2,2],[1,2,1,1,2],[1,2,1,2,1],[1,2,1,2,3],[1,2,1,
3,2],[1,2,2,1,1],[1,2,2,1,3],[1,2,2,3,1],[1,2,3,1,2],[1,2,3,2,
1],[1,3,1,2,2],[1,3,2,1,2],[1,3,2,2,1],[2,1,1,1,2],[2,1,1,2,1]
,[2,1,1,2,3],[2,1,1,3,2],[2,1,2,1,1],[2,1,2,1,3],[2,1,2,3,1],[
2,1,3,1,2],[2,1,3,2,1],[2,2,1,1,1],[2,2,1,1,3],[2,2,1,3,1],[2,
2,3,1,1],[2,3,1,1,2],[2,3,1,2,1],[2,3,2,1,1],[3,1,1,2,2],[3,1,
2,1,2],[3,1,2,2,1],[3,2,1,1,2],[3,2,1,2,1],[3,2,2,1,1]]
```

Aby otrzymać wyrazy, należy w powyższym wyniku dokonać zamiany liczby 1 na literę a, liczby 2 na literę b, itd. W funkcji warunki(x) można użyć operatorów not, and, or, xor, a także funkcji zwracających wartość typu logicznego (np. evenp(w[1]), oddp(w[1])), podobnie jak to opisano w funkcji all_solutions. Aby analitycznie wyznaczyć liczbę wyrazów, należy zastosować

wykładniczą funkcję tworzącą. W przypadku, gdy warunek logiczny nie jest koniunkcją kilku elementarnych warunków, do obliczenia liczby wyrazów należy stosować kilkakrotnie wykładniczą funkcję tworzącą i teorię liczebności zbiorów.

binary_strings(liczba_zer, liczba_jedynek) – zwraca wszystkie ciągi binarne, które można utworzyć z zadanej liczby zer i zadanej liczby jedynek. Z kombinatorycznego punktu widzenia są to permutacje z powtórzeniami (liczba_zer + liczba_jedynek)-elementowe zbioru 2-elementowego {0, 1}. Ciągów binarnych jest: $\frac{(l_{zer}+l_{jedynek})!}{l_{zer}!\,l_{jedynek}!} = \binom{l_{zer}+l_{jedynek}}{l_{zer}} = \binom{l_{zer}+l_{jedynek}}{l_{jedynek}}$.

combinations(k, n) – zwraca wszystkie kombinacje k-elementowe bez powtórzeń zbioru n-elementowego {1, 2, …, n}, których jest $\binom{n}{k}$.

all_combinations(n) – zwraca wszystkie kombinacje bez powtórzeń (wszystkie podzbiory) zbioru n-elementowego {1, 2, …, n}, których jest 2^n. Funkcja generuje wszystkie ciągi binarne n-bitowe, a następnie każdemu ciągowi przypisuje kombinację (podzbiór) na zasadzie: 0 oznacza obecność elementu w podzbiorze, natomiast 1 oznacza nieobecność elementu w podzbiorze.

all_combinations_gray(n) – zwraca wszystkie kombinacje bez powtórzeń (wszystkie podzbiory) zbioru n-elementowego {1, 2, …, n} w takiej kolejności, że każdy następny podzbiór różni się od poprzedniego tylko jednym elementem.

multicombinations(k, n) - zwraca wszystkie kombinacje k-elementowe z powtórzeniami zbioru n-elementowego {1, 2, …, n}, których jest $\binom{n+k-1}{k}$.

weak_integer_compositions(n, k) – zwraca wszystkie słabe kompozycje k-składnikowe liczby naturalnej n, których jest $\binom{n+k-1}{k-1}$.

integer_compositions(n, k) – zwraca wszystkie silne kompozycje k-składnikowe liczby naturalnej n, których jest $\binom{n-1}{k-1}$.

all_integer_compositions(n) – zwraca wszystkie silne kompozycje liczby n, których jest 2^{n-1}.

int2bin(n, ile_bitow) – konwertuje liczbę naturalną dziesiętną na liczbę w zapisie binarnym. Jeżeli potrzeba mniej bitów niż ile_bitów, wówczas z przodu wyniku dopisze zera. Jeżeli potrzeba więcej bitów niż ile_bitow, wtedy użyje minimalnej liczby, większej niż ile_bitów.

variations_with_repetitions(k, n) – generuje wszystkie wariacje k-elementowe z powtórzeniami zbioru n-elementowego {1, 2, …, n}. Liczbę takich wariacji można obliczyć ze wzoru $W_n^k = n^k$. W wariacjach kolejność elementów ma znaczenie, a więc wariacje są ciągami. Przykłady:

```
(%i14)  variations_with_repetitions(3,2);
(%o14)  [[1,1,1],[1,1,2],[1,2,1],[1,2,2],[2,1,1],[2,1,2],[2,2,1],[2,2,2]]

(%i15)  variations_with_repetitions(2,3);
(%o15)  [[1,1],[1,2],[1,3],[2,1],[2,2],[2,3],[3,1],[3,2],[3,3]]
```

variations_without_repetitions(k, n) – generuje wszystkie wariacje k-elementowe bez powtórzeń zbioru n-elementowego {1, 2, …, n}. Liczbę takich wariacji można obliczyć ze wzoru $V_n^k = \frac{n!}{(n-k)!}$. W wariacjach kolejność elementów ma znaczenie, a więc wariacje są ciągami. W języku angielski te obiekty kombinatoryczne są nazywane k-permutacjami, gdyż są podobne do permutacji, z tą różnicą, że wybiera się tylko k spośród n elementów i dla wszystkich takich wyborów tworzy się wszystkie możliwe ciągi k-elementowe. W szczególności, jeżeli $k = n$, wówczas wariacje są równoważne permutacjom zbioru {1, 2, …, n}. Przykłady:

```
(%i23)  variations_without_repetitions(2,4);
(%o23)  [[1,2],[1,3],[1,4],[2,1],[2,3],[2,4],[3,1],[3,2],[3,4],
[4,1],[4,2],[4,3]]

(%i24)  variations_without_repetitions(3,4);
(%o24)  [[1,2,3],[1,2,4],[1,3,2],[1,3,4],[1,4,2],[1,4,3],[2,1,3]
,[2,1,4],[2,3,1],[2,3,4],[2,4,1],[2,4,3],[3,1,2],[3,1,4],[3,2,1],
[3,2,4],[3,4,1],[3,4,2],[4,1,2],[4,1,3],[4,2,1],[4,2,3],[4,3,1],[
4,3,2]]

(%i25)  variations_without_repetitions(4,4);
(%o25)  [[1,2,3,4],[1,2,4,3],[1,3,2,4],[1,3,4,2],[1,4,2,3],[1,4,
3,2],[2,1,3,4],[2,1,4,3],[2,3,1,4],[2,3,4,1],[2,4,1,3],[2,4,3,1],
[3,1,2,4],[3,1,4,2],[3,2,1,4],[3,2,4,1],[3,4,1,2],[3,4,2,1],[4,1,
2,3],[4,1,3,2],[4,2,1,3],[4,2,3,1],[4,3,1,2],[4,3,2,1]]
```

permanent_by_definition(macierz) – oblicza permanent prostokątnej macierzy mxn, zgodnie z definicją. Argumentem tej funkcji może być dowolna macierz (zero-jedynkowa, macierz kosztów lub inna). Jeżeli macierz posiada więcej wierszy niż kolumn, wówczas jej permanent wynosi 0. Zgodnie z definicją, permanent macierzy mxn jest sumą tylu iloczynów, ile jest wariacji bez powtórzeń m-elementowych zbioru n-elementowego, czyli jest sumą $V_n^m = \frac{n!}{(n-m)!}$ iloczynów. Każdy iloczyn posiada m czynników, branych po jednym z każdego wiersza tak, aby z każdej kolumny był wzięty co najwyżej jeden element. Oznacza to, że n-m kolumn zostanie niewykorzystanych (nie będzie miało wybranego elementu). Nawet jeśli jakiś iloczyn zawiera zero(a) jako czynnik(i), też jest obliczany (mimo, że jest równy 0). Jest to najwolniejszy sposób obliczania permanentu. Szybkość tego sposobu nie zależy od stopnia wypełnienia macierzy zerami.

permanent_by_definition2(macierz) – oblicza permanent prostokątnej macierzy mxn zgodnie z definicją, ale w sposób bardziej optymalny. Argumentem tej funkcji może być dowolna macierz (zero-jedynkowa, macierz kosztów lub inna). Jeżeli macierz posiada więcej wierszy niż kolumn, wówczas jej permanent wynosi 0. Zgodnie z definicją, permanent macierzy mxn jest sumą tylu iloczynów, ile jest wariacji bez powtórzeń m-elementowych zbioru n-elementowego. Każdy iloczyn posiada m czynników, branych po jednym z każdego wiersza tak, aby z każdej kolumny był wzięty co najwyżej jeden element. **Ten sposób oblicza tylko niezerowe iloczyny**. Osiągnięto to w następujący sposób. Dla każdego wiersza wyznacza się zbiór numerów kolumn z niezerowymi elementami. Następnie, dla tak utworzonego ciągu zbiorów, wyznacza się wszystkie systemy różnych reprezentantów (wszystkie trans-wersale). Każdemu sytemowi reprezentantów odpo-wiada jeden niezerowy iloczyn w permanencie. Ten sposób jest szybszy niż funkcja permanent_by

_definition(macierz). Czas obliczeń tym sposobem jest tym krótszy od poprzedniego, im macierz zawiera więcej zer. Jeżeli macierz jest prawie pełna, wtedy porównaniu czasów odpowiada porównanie szybkości algorytmów generowania wariacji bez powtórzeń i generowania wszystkich transwersal dla ciągu zbiorów. Czasy te są zbliżone do siebie.

permanent_ryser(macierz) – oblicza permanent macierzy prostokątnej mxn metodą Rysera. **Macierz musi być zero-jedynkowa.** Jeżeli macierz posiada więcej wierszy niż kolumn, wówczas jej permanent wynosi 0. Ten sposób korzysta ze wzoru $per\ A = \sum_{k=0}^{m-1} (-1)^k \binom{n-m+k}{k} S_{n-m+k}(A)$, w którym $S_r(A)$ oznacza sumę iloczynów sum wierszy $\binom{n}{r}$ macierzy powstałych z oryginalnej macierzy A poprzez zastąpienie *r* jej kolumn zerami na wszystkie możliwe sposoby, których jest właśnie $\binom{n}{r}$. Numery kolumn, które należy zastąpić zerami, można otrzymać wypisując wszystkie kombinacje bez powtórzeń *r*-elementowe zbioru *n*-elementowego {1, 2, …, n}, których jest $\binom{n}{r}$. Ten sposób obliczania permanentu jest dosyć szybki.

permanent_recursively(macierz) – ten sposób obliczania permanentu stosuje rozwinięcie macierzy względem wybranego wiersza (metoda rozwinięcia Laplace'a). Nie można jednak przekształcać macierzy tak, aby miała w wybranym wierszu czy kolumnie tylko jeden niezerowy element. Takich twierdzeń, jak dla wyznacznika macierzy, nie ma dla permanenetu. Ten sposób jest szybki i można go stosować dla każdej macierzy, tzn. o dowolnych wartościach i rozmiarach.

random_matrix01(m, n, [p]) – zwraca losową zero-jedynkową macierz mxn, w której prawdopodobieństwo wystąpienia jedynki wynosi 1/2, gdy nie użyto trzeciego argumentu. Aby zmienić domyślne prawdopodobieństwo wystąpienia jedynki, należy podać trzeci argument (liczbę rzeczywistą z przedziału <0.0 do 1.0>).

random_cost_matrix(m,n,wartosc_min,wartosc_maks, p_zera, [wartosc_przekatnej]) – zwraca losową macierz kosztów o rozmiarach mxn, z wartosciami losowanymi równomiernie z przedziału (wartosc_min, wartosc_maks). Prawdopodobieństwo wylosowania zera definiuje zmienna p_zero o wartościach z przedziału <0.0, 1.0>. Można użyć opcjonalnego argumentu, który ustawi wartosci wszystkich komórek macierzy o współrzędnych [i,i] na taką samą wartość. Jeżeli użyć jeszcze jednego opcjonalnego argumentu (dowolnego), wtedy macierz będzie symetryczna. Ta opcja działa tylko wtedy, gdy macierz jest kwadratowa, a więc dla m=n. Dzięki tej opcji możemy wygenerować macierz wag grafu nieskierowanego (macierz kwadratowa, symetryczna, z zerami na przekątnej) albo skierowanego (macierz kwadratowa, niesymetryczna, z zerami na przekątnej). Macierz prostokątna przydaje się do modelowania problemu optymalnego przydziału np. pracowników do prac.

```
(%i50)  macierz : random_cost_matrix(4, 6, 5, 15, 0.3);
```

(macierz)

$$\begin{bmatrix} 12 & 9 & 14 & 12 & 10 & 13 \\ 10 & 0 & 0 & 8 & 8 & 6 \\ 7 & 11 & 0 & 0 & 11 & 6 \\ 0 & 14 & 0 & 0 & 12 & 5 \end{bmatrix}$$

```
(%i55)  macierz : random_cost_matrix(4, 6, 5, 5, 0.3);
```

(macierz)

$$\begin{bmatrix} 5 & 5 & 5 & 5 & 0 & 5 \\ 5 & 5 & 5 & 5 & 5 & 5 \\ 5 & 5 & 0 & 0 & 5 & 5 \\ 5 & 0 & 5 & 5 & 5 & 0 \end{bmatrix}$$

```
(%i57)  macierz : random_cost_matrix(5, 5, 10, 20, 0.2, 0);
```

(macierz)

$$\begin{bmatrix} 0 & 12 & 0 & 10 & 10 \\ 13 & 0 & 13 & 19 & 18 \\ 10 & 0 & 0 & 19 & 0 \\ 0 & 13 & 10 & 0 & 12 \\ 0 & 16 & 14 & 11 & 0 \end{bmatrix}$$

```
(%i67)  macierz : random_cost_matrix(5, 5, 10, 20, 0.2, 0, symetryczna);
```

(macierz)

$$\begin{bmatrix} 0 & 17 & 13 & 0 & 11 \\ 17 & 0 & 0 & 16 & 0 \\ 13 & 0 & 0 & 15 & 15 \\ 0 & 16 & 15 & 0 & 11 \\ 11 & 0 & 15 & 11 & 0 \end{bmatrix}$$

matrix_to_bipartite_graph(macierz, czy_wagi) – konwertuje macierz (kosztów) do odpowiadającego jej nieskierowanego grafu dwudzielnego (V1,V2). Zbiory wierzchołków V1 i V2 posiadają, odpowiednio, tyle wierzchołków, ile wierszy i kolumn ma macierz. Elementowi a[i,j] różnemu od zera odpowiada krawędź [i, j+m], gdzie m to liczba wierszy macierzy. Zmienna boolowska czy_wagi decyduje o tym, czy graf ma być z wagami a[i,j], czy bez wag (tylko krawędzie). Przykład:

```
(%i97)  macierz : random_cost_matrix(3, 5, 6, 28, 0.2);
        graf_bez_wag  : matrix_to_bipartite_graph(macierz, false);
        graf_z_wagami : matrix_to_bipartite_graph(macierz, true);
```

(macierz)
$$\begin{bmatrix} 25 & 0 & 19 & 8 & 26 \\ 6 & 19 & 22 & 0 & 18 \\ 12 & 0 & 15 & 18 & 0 \end{bmatrix}$$

(graf_bez_wag) [[1,4],[1,6],[1,7],[1,8],[2,4],[2,5],[2,6],[2,8],[3,4],[3,6],[3,7]]

(graf_z_wagami) [[[1,4],25],[[1,6],19],[[1,7],8],[[1,8],26],[[2,4],6],[[2,5],19],[[2,6],22],[[2,8],18],[[3,4],12],[[3,6],15],[[3,7],18]]

matrix_to_digraph(macierz, czy_wagi) – konwertuje kwadratową macierz (kosztów), niekoniecznie symetryczną, na graf skierowany, tzn. dla każdego elementu a[i,j]#0 i nie leżącego na przekątnej (pętle w digrafie są pomijane) dodaje jeden łuk do digrafu. Wagi można zignorować (pominąć), jeżeli nie są potrzebne w digrafie. Przykład użycia tej funkcji:

```
(%i123) macierz : random_cost_matrix(5, 5, 20, 30, 0.35, 0);
        digraf_bez_wag : matrix_to_digraph(macierz, false);
        digraf_z_wagami : matrix_to_digraph(macierz,true);
```

(macierz)
$$\begin{bmatrix} 0 & 23 & 26 & 26 & 0 \\ 29 & 0 & 0 & 28 & 29 \\ 0 & 0 & 0 & 28 & 28 \\ 21 & 23 & 22 & 0 & 0 \\ 20 & 29 & 22 & 27 & 0 \end{bmatrix}$$

(digraf_bez_wag) [[1,2],[1,3],[1,4],[2,1],[2,4],[2,5],[3,4],[3,5],[4,1],[4,2],[4,3],[5,1],[5,2],[5,3],[5,4]]

(digraf_z_wagami) [[[1,2],23],[[1,3],26],[[1,4],26],[[2,1],29],[[2,4],28],[[2,5],29],[[3,4],28],[[3,5],28],[[4,1],21],[[4,2],23],[[4,3],22],[[5,1],20],[[5,2],29],[[5,3],22],[[5,4],27]]

```
(%i132)  macierz : random_cost_matrix(5, 5, 20, 30, 0.3, 0, sym);
         digraf_bez_wag : matrix_to_digraph(macierz, false);
         digraf_z_wagami : matrix_to_digraph(macierz,true);
```

(macierz)
$$\begin{bmatrix} 0 & 22 & 21 & 0 & 23 \\ 22 & 0 & 24 & 20 & 20 \\ 21 & 24 & 0 & 21 & 23 \\ 0 & 20 & 21 & 0 & 0 \\ 23 & 20 & 23 & 0 & 0 \end{bmatrix}$$

(digraf_bez_wag) [[1,2],[1,3],[1,5],[2,1],[2,3],[2,4],[2,5],[3,1],[3,2],[3,4],[3,5],[4,2],[4,3],[5,1],[5,2],[5,3]]

(digraf_z_wagami) [[[1,2],22],[[1,3],21],[[1,5],23],[[2,1],22],[[2,3],24],[[2,4],20],[[2,5],20],[[3,1],21],[[3,2],24],[[3,4],21],[[3,5],23],[[4,2],20],[[4,3],21],[[5,1],23],[[5,2],20],[[5,3],23]]

cost_matrix_to_sets(macierz, sortuj) – dla zadanej macierzy kosztów wyznacza zbiory zawierające numery niezerowych kolumn w poszczególnych wierszach. Jeżeli sortuj jest równe 0, wtedy zbiory nie są sortowane. Jeżeli sortuj wynosi +1 (-1), wtedy numery kolumn w zbiorach są sortowane rosnąco (malejąco) ze względu na odpowiadające im wagi w danym wierszu macierzy. Zbiory niezerowych kolumn generuje się w celu wyznaczenia dla nich wszystkich systemów różnych reprezentantów (transwersal). Transwersala, w zależności od zadania, którego dotyczy rozważana macierz, może kodować dykl Hamiltona w digrafie lub jeden z wielu przydziałów pracowników do prac. Obliczają sumę wartości elementów macierzy kosztów, odpowiadających danej transwersali, można znaleźć koszt pracy całego zespołu. Przykłady użycia:

```
(%i174)  macierz : random_cost_matrix(4, 6, 10, 20, 0.3);
         zbiory_bez_sortowania : cost_matrix_to_sets(macierz, 0);
         zbiory_sortowane_rosnaco : cost_matrix_to_sets(macierz, 1);
         zbiory_sortowane_malejaco : cost_matrix_to_sets(macierz,-1);
```

(macierz)
$$\begin{bmatrix} 12 & 16 & 0 & 10 & 0 & 14 \\ 0 & 18 & 16 & 13 & 0 & 10 \\ 15 & 0 & 14 & 11 & 12 & 16 \\ 0 & 19 & 0 & 12 & 16 & 16 \end{bmatrix}$$

(zbiory_bez_sortowania) [[1,2,4,6],[2,3,4,6],[1,3,4,5,6],[2,4,5,6]]

(zbiory_sortowane_rosnaco) [[4,1,6,2],[6,4,3,2],[4,5,3,1,6],[4,5,6,2]]

(zbiory_sortowane_malejaco) [[2,6,1,4],[2,3,4,6],[6,1,3,5,4],[2,5,6,4]]

optimal_assignment(macierz_kosztow, sortuj, [wykres]) – funkcja rozwiązuje problem optymalnego przydziału, opisany macierzą kosztów, nie metodą algorytmu węgierskiego, lecz metodą transwersal, czyli przeglądnięcia wszyst-

kich możliwych przydziałów pracowników do prac. Jeżeli sortuj jest równe 0, wtedy zbiory numerów niezerowych kolumn w wierszach nie są sortowane. Jeżeli sortuj wynosi +1 (-1), wtedy te zbiory są sortowane rosnąco (malejąco) ze względu na wagi w danym wierszu macierzy. Sortowanie numerów kolumn w zbiorach pozwala przesunąć optymalny (minimalny albo maksymalny) przydział bliżej początku listy transwersal. Jeżeli funkcja optimal_assignment zostanie wywołana tylko z argumentami macierz_kosztow i sortuj, wtedy zwróci tylko minimalny i maksymalny przydział pracowników do prac. Jeżeli użyć jednego opcjonalnego argumentu, wtedy funkcja zwróci wszystkie przydziały, wśród których będą też te oba ekstremalne. Jeżeli użyć dwóch opcjonalnych argumentów, wtedy dodatkowo zostanie sporządzany wykres ilustrujący rozwiązanie optymalnego przydziału dla zadanej macierzy. Jeżeli macierz kosztów jest kwadratowa i posiada zera na przekątnej, wówczas jest traktowana jak macierz wag łuków grafu skierowanego. W tym przypadku funkcja optimal_assignment sprawdzi, które transwersale kodują cykle Hamiltona w digrafie i wyznaczy wszystkie cykle wraz z sumą wag łuków. Wyznaczy także wszystkie pozostałe przydziały, gdyż można tę macierz traktować jak zwykłą macierz kosztów, a nie macierz wag digrafu.

```
(%i220) macierz : random_cost_matrix(4, 6, 50, 99, 0.25);
        optimal_assignment(macierz,0);
```

(macierz)

$$\begin{bmatrix} 52 & 63 & 96 & 60 & 92 & 71 \\ 0 & 95 & 88 & 53 & 0 & 90 \\ 56 & 0 & 65 & 53 & 69 & 57 \\ 54 & 72 & 0 & 98 & 0 & 82 \end{bmatrix}$$

```
(%o220) [[[2,4,6,1],[63,53,57,54],227],[[3,2,5,4],[96,95,69,98],358]]

(%i221) optimal_assignment(macierz,1);
(%o221) [[[2,4,6,1],[63,53,57,54],227],[[3,2,5,4],[96,95,69,98],358]]

(%i222) optimal_assignment(macierz,-1);
(%o222) [[[2,4,6,1],[63,53,57,54],227],[[3,2,5,4],[96,95,69,98],358]]
--------------------------------------------------
```

```
(%i258)  macierz : random_cost_matrix(3, 5, 40, 60, 0.15);
         optimal_assignment(macierz,0);
```

(macierz)

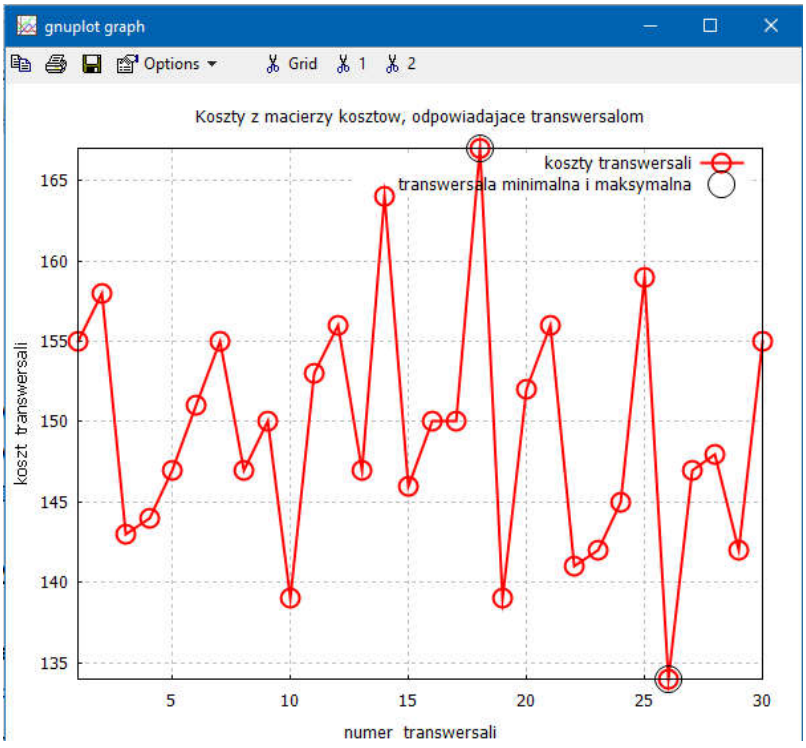
$$\begin{bmatrix} 43 & 52 & 0 & 52 & 47 \\ 41 & 58 & 47 & 55 & 0 \\ 40 & 53 & 0 & 54 & 57 \end{bmatrix}$$

```
(%o258)  [[[5,3,1],[47,47,40],134],[[4,2,5],[52,58,57],167]]

(%i261)  optimal_assignment(macierz,0,pokaz);
(%o261)  [[[1,2,4],155],[[1,2,5],158],[[1,3,2],143],[[1,3,4],144],[[1,3,5],
147],[[1,4,2],151],[[1,4,5],155],[[2,1,4],147],[[2,1,5],150],[[2,3,1],139],[
[2,3,4],153],[[2,3,5],156],[[2,4,1],147],[[2,4,5],164],[[4,1,2],146],[[4,1,5
],150],[[4,2,1],150],[[4,2,5],167],[[4,3,1],139],[[4,3,2],152],[[4,3,5],156
],[[5,1,2],141],[[5,1,4],142],[[5,2,1],145],[[5,2,4],159],[[5,3,1],134],[[5,
3,2],147],[[5,3,4],148],[[5,4,1],142],[[5,4,2],155]]
(%i262)  optimal_assignment(macierz,1,pokaz);
(%o262)  [[[1,3,2],143],[[1,3,4],144],[[1,3,5],147],[[1,4,2],151],[[1,4,5],
155],[[1,2,4],155],[[1,2,5],158],[[5,1,2],141],[[5,1,4],142],[[5,3,1],134],[
[5,3,2],147],[[5,3,4],148],[[5,4,1],142],[[5,4,2],155],[[5,2,1],145],[[5,2,4
],159],[[2,1,4],147],[[2,1,5],150],[[2,3,1],139],[[2,3,4],153],[[2,3,5],156
],[[2,4,1],147],[[2,4,5],164],[[4,1,2],146],[[4,1,5],150],[[4,3,1],139],[[4,
3,2],152],[[4,3,5],156],[[4,2,1],150],[[4,2,5],167]]

(%i263)  optimal_assignment(macierz,-1,pokaz);
(%o263)  [[[2,4,5],164],[[2,4,1],147],[[2,3,5],156],[[2,3,4],153],[[2,3,1],
139],[[2,1,5],150],[[2,1,4],147],[[4,2,5],167],[[4,2,1],150],[[4,3,5],156],[
[4,3,2],152],[[4,3,1],139],[[4,1,5],150],[[4,1,2],146],[[5,2,4],159],[[5,2,1
],145],[[5,4,2],155],[[5,4,1],142],[[5,3,4],148],[[5,3,2],147],[[5,3,1],134
],[[5,1,4],142],[[5,1,2],141],[[1,2,5],158],[[1,2,4],155],[[1,4,5],155],[[1,
4,2],151],[[1,3,5],147],[[1,3,4],144],[[1,3,2],143]]

(%i265)  optimal_assignment(macierz,0,pokaz, wykres);
```

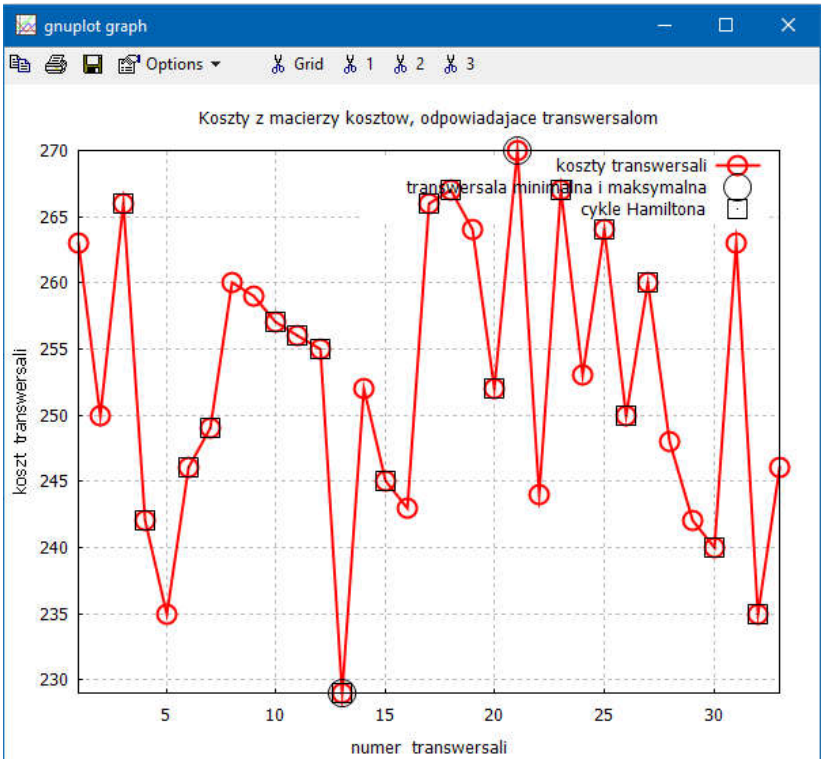


```
(%i291)  macierz : random_cost_matrix(5, 5, 40, 60, 0.1, 0);
         oa : optimal_assignment(macierz,0,pokaz,wykres)$
         cykle_hamiltona : oa[2];
```

$$(\text{macierz}) \begin{bmatrix} 0 & 50 & 53 & 52 & 43 \\ 52 & 0 & 57 & 41 & 50 \\ 0 & 53 & 0 & 51 & 46 \\ 43 & 59 & 56 & 0 & 55 \\ 53 & 46 & 55 & 46 & 0 \end{bmatrix}$$

0 errors, 0 warnings

(cykle_hamiltona) ***[[[2,3,4,5,1],266],[[2,3,5,1,4],242],[[2,4,5,3,1],246],[[2,5,4,1,3],249],[[3,1,4,5,2],257],[[3,1,5,2,4],256],[[3,4,2,5,1],255],[[3,4,5,1,2],229],[[3,5,2,1,4],245],[[3,5,4,2,1],266],[[4,1,2,5,3],267],[[4,1,5,3,2],252],[[4,3,5,2,1],267],[[4,5,2,3,1],264],[[5,1,2,3,4],250],[[5,1,4,2,3],260],[[5,3,4,1,2],240],[[5,4,2,1,3],235]]***



`all_transversals(zbiory,[ile])` – funkcja ta generuje wszystkie systemy różnych reprezentantów dla podanego ciągu zbiorów. W praktycznych przypadkach te zbiory są numerami kolumn z niezerowymi kosztami w danym wierszu macierzy kosztów. Opcjonalny argument ile określa, po ilu wygenerowanych transwersalach funkcja ma zakończyć działanie. Ta opcja jest przydatna wówczas, gdy wszystkich transwersal jest dużo, a nas interesuje np. tylko to, czy jest co najmniej jedna. Taka transwersala może kodować np. skojarzenie pełne w grafie dwudzielnym, zakodowanym podanym ciągiem zbiorów.

Jeżeli argument ile jest liczbą całkowitą <= 0, wtedy zostaną wygenerowane wszystkie transwersale, czyli ta funkcja działa tak, jak wywołana z jednym argumentem. Przykład:

```
(%i366) macierz : random_cost_matrix(4, 5, 20, 40, 0.33);
        zbiory : cost_matrix_to_sets(macierz,0);
        transwersale : all_transversals(zbiory);
```

(macierz)
$$\begin{bmatrix} 28 & 27 & 21 & 21 & 0 \\ 22 & 21 & 0 & 0 & 29 \\ 24 & 20 & 30 & 36 & 0 \\ 0 & 24 & 0 & 39 & 0 \end{bmatrix}$$

(zbiory) [[1,2,3,4],[1,2,5],[1,2,3,4],[2,4]]

(transwersale) [[1,2,3,4],[1,5,2,4],[1,5,3,2],[1,5,3,4],[1,5,4,2],[2,1,3,4],[2,5,1,4],[2,5,3,4],[3,1,2,4],[3,1,4,2],[3,2,1,4],[3,5,1,2],[3,5,1,4],[3,5,2,4],[3,5,4,2],[4,1,3,2],[4,5,1,2],[4,5,3,2]]

```
(%i369) trzy_pierwsze : all_transversals(zbiory,3);
```

(trzy_pierwsze) [[1,2,3,4],[1,5,2,4],[1,5,3,2]]

sets_to_matrix01(zbiory) – konwertuje ciąg zbiorów, których elementy są rozumiane jako numery niezerowych kolumn w kolejnych wierszach, na macierz zero-jedynkową. Przykład:

```
(%i15) sets_to_matrix01( [ [2,1], [3,1,5], [5,2,4] ]);
```

(%o15)
$$\begin{bmatrix} 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \end{bmatrix}$$

matrix01_to_sets(macierz) – konwertuje macierz zero-jedynkową na ciąg zbiorów zawierających numery niezerowych kolumn w kolejnych wierszach tej macierzy. Przykład:

```
(%i20) macierz : matrix([1,1,0,0,0], [1,0,1,0,1], [0,1,0,1,1]);
       matrix01_to_sets(macierz);
```

(macierz)
$$\begin{bmatrix} 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \end{bmatrix}$$

(%o20) [[1,2],[1,3,5],[2,4,5]]

hall_theorem(zbiory, [wykres]) – dla zadanego ciągu zbiorów funkcja sprawdza, czy spełnione jest tw. Halla o istnieniu pełnego skojarzenia w grafie dwudzielnym. W tym przypadku każdy zbiór definiuje sąsiadów dla kolejnych wierzchołków zbioru V1 grafu dwudzielnego. Twierdzenie to może również dotyczyć macierzy. Wtedy zbiory zawierają numery niezerowych kolumn w kolejnych wierszach macierzy. Jeżeli wywołać tę funkcję z dwoma argumentami, to zostanie sporządzony wykres liczebności zbioru sąsiadów

danego podzbioru w funkcji liczebności podzbioru.

Przykłady:

```
(%i692)  macierz : random_matrix01(3,5,0.5);
         zbiory : matrix01_to_sets(macierz);
         podzbior_sasiedzi : hall_theorem(zbiory, wykres);
         liczebnosci : map( lambda( [para], [length(para[1]), length(para[2])] ), podzbior_sasiedzi);
         nierownosci : map( lambda( [para], [ is(para[1]<=para[2]) ] ), liczebnosci);
```

(macierz)
$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 \end{bmatrix}$$

(zbiory) *[[4],[1,4],[2,4,5]]*

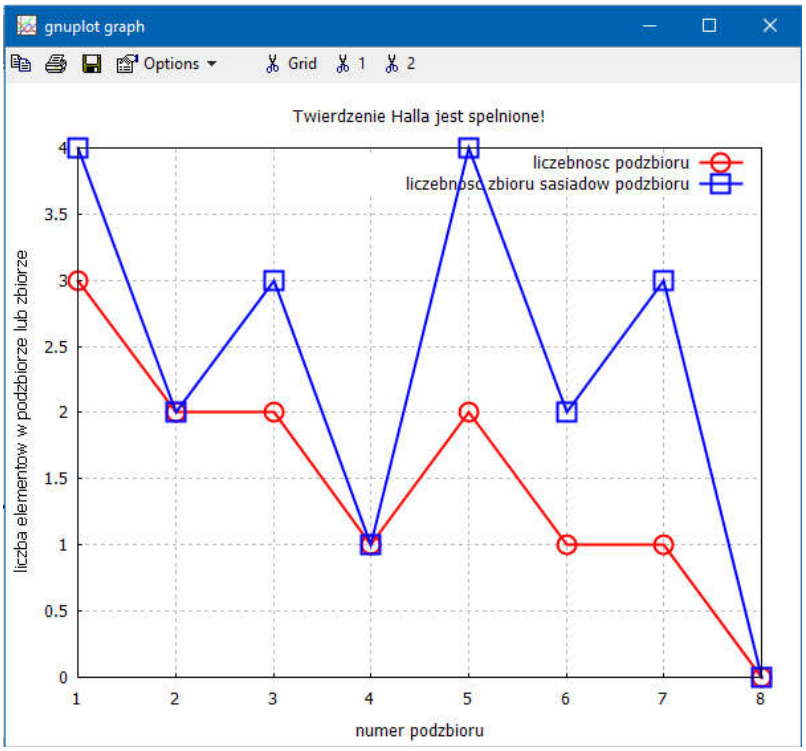
Twierdzenie Halla jest spelnione!

0 errors, 0 warnings

(podzbior_sasiedzi) *[[[1,2,3],[1,2,4,5]],[[1,2],[1,4]],[[1,3],[2,4,5]],[[1],[4]],[[2,3],[1,2,4,5]],[[2],[1,4]],[[3],[2,4,5]],[[],[]]]*

(liczebnosci) *[[3,4],[2,2],[2,3],[1,1],[2,4],[1,2],[1,3],[0,0]]*

(nierownosci) ***[[true],[true],[true],[true],[true],[true],[true],[true]]***



```
(%i722)  macierz : random_matrix01(3,5,0.3);
         zbiory : matrix01_to_sets(macierz);
         podzbior_sasiedzi : hall_theorem(zbiory, wykres);
         liczebnosci : map( lambda( [para], [length(para[1]), length(para[2])] ), podzbior_sasiedzi);
         nierownosci : map( lambda( [para], [ is(para[1]<=para[2]) ] ), liczebnosci);
```

(macierz)
$$\begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

(zbiory) *[[5],[5],[3,4]]*

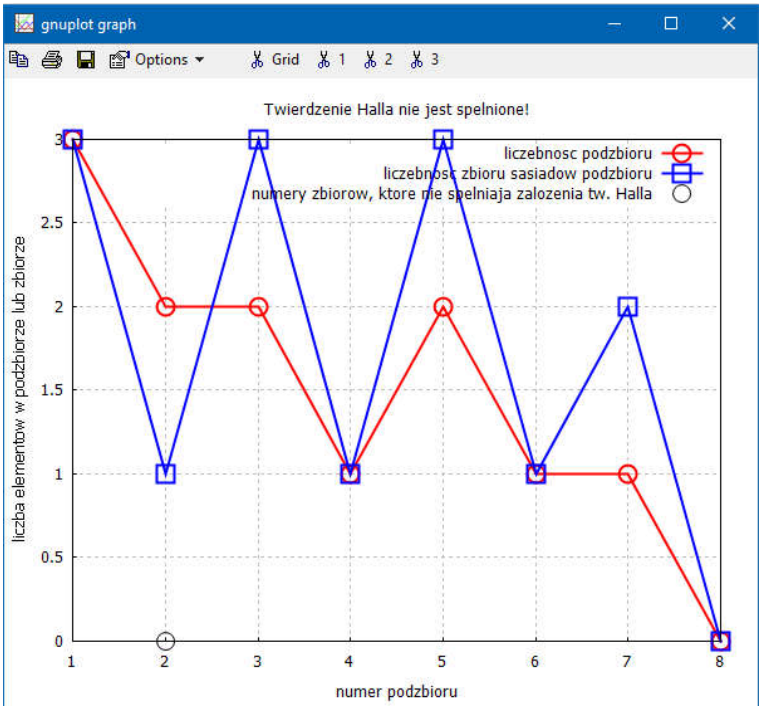
Twierdzenie Halla nie jest spelnione!

0 errors, 0 warnings

(podzbior_sasiedzi) *[[[1,2,3],[3,4,5]],[[1,2],[5]],[[1,3],[3,4,5]],[[1],[5]],[[2,3],[3,4,5]],[[2],[5]],[[3],[3,4]],[[],[]]]*

(liczebnosci) *[[3,3],[2,1],[2,3],[1,1],[2,3],[1,1],[1,2],[0,0]]*

(nierownosci) ***[[true],[false],[true],[true],[true],[true],[true],[true]]***



all_maximal_independent_vertex_sets(graf, [licznosc]) – wyznacza wszystkie maksymalne zbiory niezależne wierzchołków w grafie nieskierowanym. Jeżeli użyć opcjonalnego argumentu liczność (liczby całkowitej), to: dla liczność=0 funkcja zwróci tylko największe zbiory maksymalne; dla innych wartości zmiennej liczność funkcja zwróci zbiory maksymalne, które posiadają tyle wierzchołków, ile wynosi wartość zmiennej liczność. Przykłady:

```
(%i738)  graf : cycle_Graph(6);
         wszystkie : all_maximal_independent_vertex_sets(graf);
         najwieksze : all_maximal_independent_vertex_sets(graf,0);
         jedno_wierzcholkowe : all_maximal_independent_vertex_sets(graf,1);
         dwu_wierzcholkowe : all_maximal_independent_vertex_sets(graf,2);
(graf)   [[1,2],[2,3],[3,4],[4,5],[5,6],[6,1]]
(wszystkie)  [[1,4],[2,5],[3,6],[1,3,5],[2,4,6]]
(najwieksze)  [[1,3,5],[2,4,6]]
(jedno_wierzcholkowe)  []
(dwu_wierzcholkowe)  [[1,4],[2,5],[3,6]]
```

all_maximal_independent_edge_sets(graf, [licznosc]) – wyznacza wszystkie maksymalne zbiory niezależne krawędzi, czyli maksymalne skojarzenia, w grafie nieskierowanym. Jeżeli użyć opcjonalnego argumentu liczność (liczby całkowitej), to: dla liczność=0 funkcja zwróci tylko największe skojarzenia; dla innych

wartości zmiennej liczność funkcja zwróci maksymalne skojarzenia, które posiadają tyle krawędzi, ile wynosi wartość zmiennej liczność. Przykłady:

```
(%i743)  graf : cycle_Graph(6);
         wszystkie : all_maximal_independent_edge_sets(graf);
         najwieksze : all_maximal_independent_edge_sets(graf,0);
         jedno_krawedziowe : all_maximal_independent_edge_sets(graf,1);
         dwu_krawedziowe : all_maximal_independent_edge_sets(graf,2);
(graf)   [[1,2],[2,3],[3,4],[4,5],[5,6],[6,1]]
(wszystkie) [[[1,2],[4,5]],[[1,6],[3,4]],[[2,3],[5,6]],[[1,2],[3,4],[5,6]],
[[1,6],[2,3],[4,5]]]
(najwieksze) [[[1,2],[3,4],[5,6]],[[1,6],[2,3],[4,5]]]
(jedno_krawedziowe) []
(dwu_krawedziowe) [[[1,2],[4,5]],[[1,6],[3,4]],[[2,3],[5,6]]]
```

all_minimal_vertex_covers(graf, [licznosc]) – wyznacza wszystkie minimalne pokrycia wierzchołkowe (bazy minimalne) grafu nieskierowanego. Jeżeli użyć opcjonalnego argumentu liczność (liczby całkowitej), to: dla liczność=0 funkcja zwróci tylko najmniejsze minimalne pokrycia wierzchołkowe; dla innych wartości zmiennej liczność funkcja zwróci minimalne pokrycia, które posiadają tyle wierzchołków, ile wynosi wartość zmiennej liczność. Przykłady:

```
(%i768)  graf : path_Graph(5);
         wszystkie : all_minimal_vertex_covers(graf);
         najmniejsze : all_minimal_vertex_covers(graf,0);
         trzy_wierzcholkowe : all_minimal_vertex_covers(graf,3);
         cztero_wierzcholkowe : all_minimal_vertex_covers(graf,4);
(graf)   [[1,2],[2,3],[3,4],[4,5]]
(wszystkie) [[2,4],[1,3,4],[1,3,5],[2,3,5]]
(najmniejsze) [[2,4]]
(trzy_wierzcholkowe) [[1,3,4],[1,3,5],[2,3,5]]
(cztero_wierzcholkowe) []
```

all_minimal_edge_covers(graf, [licznosc]) – wyznacza wszystkie minimalne pokrycia krawędziowe w grafie niekierowanym. Jeżeli użyć opcjonalnego argumentu liczność (liczby całkowitej), to: dla liczność=0 funkcja zwróci tylko najmniejsze minimalne pokrycia krawędziowe; dla innych wartości zmiennej liczność funkcja zwróci minimalne pokrycia, które posiadają tyle krawędzi, ile wynosi wartość zmiennej liczność. Przykłady:

```
(%i788)  graf : path_Graph(6);
         wszystkie : all_minimal_edge_covers(graf);
         najmniejsze : all_minimal_edge_covers(graf,0);
         trzy_krawedziowe : all_minimal_edge_covers(graf,3);
         cztero_krawedziowe : all_minimal_edge_covers(graf,4);
(graf)   [[1,2],[2,3],[3,4],[4,5],[5,6]]
(wszystkie)  [[[1,2],[3,4],[5,6]],[[1,2],[2,3],[4,5],[5,6]]]
(najmniejsze)[[[1,2],[3,4],[5,6]]]
(trzy_krawedziowe)  [[[1,2],[3,4],[5,6]]]
(cztero_krawedziowe) [[[1,2],[2,3],[4,5],[5,6]]]
```

all_minimal_dominating_vertex_sets(graf, [licznosc]) – wyznacza wszystkie minimalne zbiory dominujące wierzchołków w grafie nieskierowanym. Jeżeli użyć opcjonalnego argumentu liczność (liczby całkowitej), to: dla liczność=0 funkcja zwróci tylko najmniejsze minimalne zbiory dominujące wierzchołkowe; dla innych wartości zmiennej liczność funkcja zwróci minimalne zbiory dominujące, które posiadają tyle wierzchołków, ile wynosi wartość zmiennej liczność. Przykłady:

```
(%i858)  graf : [[1,2],[2,3],[2,4],[3,4],[3,5]];
         wszystkie : all_minimal_dominating_vertex_sets(graf);
         najmniejsze : all_minimal_dominating_vertex_sets(graf,0);
         trzy_wierzcholkowe : all_minimal_dominating_vertex_sets(graf,3);
         cztero_wierzcholkowe : all_minimal_dominating_vertex_sets(graf,4);
(graf)   [[1,2],[2,3],[2,4],[3,4],[3,5]]
(wszystkie)  [[1,3],[2,3],[2,5],[1,4,5]]
(najmniejsze)[[1,3],[2,3],[2,5]]
(trzy_wierzcholkowe)  [[1,4,5]]
(cztero_wierzcholkowe)  []
```

minimum_spanning_tree_kruskal(graf_wazony, [pokaz]) – wyznacza minimalne drzewo rozpinające grafu nieskierowanego z wagami algorytmem Kruskala. Aby zobaczyć kolejne kroki algorytmu, należy wywołać tę funkcję z dwoma

argumentami (drugi dowolny). Przykład:

```
(%i1008) graf_wazony : random_weighted_graph(5, 1.0, 10, 20);
         minimum_spanning_tree_kruskal(graf_wazony, pokaz);
(graf_wazony) [[[1,2],10],[[1,3],17],[[1,4],14],[[1,5],13],[[2,3],16],[[
2,4],16],[[2,5],12],[[3,4],19],[[3,5],19],[[4,5],15]]
Krawedzie grafu posortowane rosnaco (niemalejaco) wzgledem wag:
[[[1,2],10],[[2,5],12],[[1,5],13],[[1,4],14],[[4,5],15],[[2,3],16]
,[[2,4],16],[[1,3],17],[[3,4],19],[[3,5],19]]
```

Krawedzie drzewa	Zbiory wierzcholkow
------	[[1],[2],[3],[4],[5]]
[[1,2],10]	[[1,2],[3],[4],[5]]
[[2,5],12]	[[1,2,5],[3],[4]]
[[1,4],14]	[[1,2,5,4],[3]]
[[2,3],16]	[[1,2,5,4,3]]

```
(%o1008) [[[[1,2],10],[[2,5],12],[[1,4],14],[[2,3],16]],52]
```

minimum_spanning_tree_prim(graf_wazony, [zmienne]) - wyznacza minimalne drzewo rozpinające grafu nieskierowanego z wagami algorytmem Prima, rozpoczynając od wierzchołka nr 1. Aby rozpocząć od innego wierzchołka, należy go podać jako drugi argument tej funkcji. Jeżeli podać trzeci, dowolny argument, wówczas zostaną pokazane szczegóły działania algorytmu Prima.

```
(%i1012) graf_wazony : [[[1,2],10],[[1,3],17],[[1,4],14],[[1,5],13],[[2,3],16],
           [[2,4],16],[[2,5],12],[[3,4],19],[[3,5],19],[[4,5],15]];
         mst : minimum_spanning_tree_prim(graf_wazony, 3, pokaz);
(graf_wazony) [[[1,2],10],[[1,3],17],[[1,4],14],[[1,5],13],[[2,3],16],[[2,4],16],[[2,5],
12],[[3,4],19],[[3,5],19],[[4,5],15]]
```

Zbior S	Zbior V\S	Przekroj (S, V\S)	Krawedz lekka
[3]	[1,2,4,5]	[[[3,1],17],[[3,2],16],[[3,4],19],[[3,5],19]]	[[3,2],16]
[3,2]	[1,4,5]	[[[3,1],17],[[3,4],19],[[3,5],19],[[2,1],10],[[2,4],16],[[2,5],12]]	[[2,1],10]
[3,2,1]	[4,5]	[[[3,4],19],[[3,5],19],[[2,4],16],[[2,5],12],[[1,4],14],[[1,5],13]]	[[2,5],12]
[3,2,1,5]	[4]	[[[3,4],19],[[2,4],16],[[1,4],14],[[5,4],15]]	[[1,4],14]

```
(mst) [[[[3,2],16],[[2,1],10],[[2,5],12],[[1,4],14]],52]
```

all_hamilton_cycles_by_transversals(digraf, sortuj_luki, ile_cykli, filter) - wyznacza wszystkie skierowane cykle Hamiltona dla podanego digrafu (ważonego lub nie). Jeżeli digraf nie jest zapisany w formie digrafu, ale znana jest jego macierz wag łuków, to należy ją najpierw samodzielnie przekonwertować do digrafu. Jeżeli chcemy wyznaczyć cykle w grafie nieskierowanym, to należy go wcześniej przekonwertować do digrafu funkcją graph_to_ digraph(). Wówczas każda krawędź zostanie zastąpiona parą łuków. Opisywana funkcja wyznacza cykle Hamiltona poprzez generowanie transwersal (systemów różnych reprezentantów) dla tylu zbiorów, ile jest wierzchołków w digrafie. Każdy zbiór to lista

wierzchołków, do których łuki z danego wierzchołka bezpośrednio prowadzą. Jeżeli sortuj_luki jest >0 (<0), wtedy numery wierzchołków-sąsiadów zostaną posortowane rosnąco (malejąco) ze względu na wartości wag łuków do nich prowadzących. Jeżeli sortuj_luki jest równe 0, wtedy wierzchołki-sąsiedzi, a więc łuki, zostaną w każdym zbiorze posortowane rosnąco, ze względu na numer wierzchołka, do którego prowadzą. Argument ile_cykli pozwala zdecydować, ile cykli Hamiltona ma funkcja wyznaczyć (nie muszą nas interesować wszystkie, bo ich obliczenie może trwać zbyt długo). Jeżeli ile_cykli jest <=0 lub większe od (nieznanej) liczby wszystkich cykli Hamiltona w rozważanym digrafie, wtedy zostaną wygenerowane wszystkie cykle. Jeżeli ile_cykli jest liczbą całkowitą dodatnią i niezbyt dużą, wówczas zostanie wygenerowanych tylko tyle początkowych cykli Hamiltona, ile zażądano. Kolejność generowanych cykli zależy bowiem od sposobu posortowania numerów wierzchołków w zbiorach użytych do generowania transwersal. Jeżeli argument filter jest równy 0, wtedy funkcja zwróci tylko listę cykli Hamiltona, wraz z ich wagami. Jeżeli filter równa się 1, wtedy funkcja zwróci listę 2-elementową [dobre_transwersale, cykle_hamiltona], która zawierać będzie dwie listy: listę dobrych transwersal, czyli tych, które kodują cykle Hamiltona, oraz listę cykli Hamiltona, które tym transwersalom odpowiadają. Jeżeli filter równa się 2, wtedy funkcja zwróci wszystkie transwersale i jeżeli dana transwersalna nie koduje cyklu Hamiltona, to zostanie przypisana jej wartość -1, a jeżeli koduje, to waga cyklu. W przypadku grafu nieskierowanego każdy cykl nieskierowany pojawi się na liście cykli Hamiltona dwa razy, tzn. w kierunku (nazwijmy go) pierwotnym i odwrotnym, jeżeli oczywiście wygenerujemy wszystkie cykle.

Transwersalna koduje cykl Hamiltona, jeżeli jest permutacją jednocyklową. Ponieważ macierz wag

łuków digrafu jest kwadratowa, to wszystkie transwersale są (dokładnie) permutacjami (a nie wariacjami bez powtórzeń). Pozostaje więc tylko sprawdzić, czy permutacja jest jednocyklowa. Jeżeli zapisać wierzchołki cyklu Hamiltona w kolejności ich odwiedzenia i nie zapisać wierzchołka startowego drugi raz na końcu, wtedy powstanie ciąg, w którym każda liczba jest unikalna, czyli jest to permutacja 1-cyklowa.

Uwaga: wierzchołki digrafu muszą być ponumerowane od 1. Wagi łuków mogą być liczbami rzeczywistymi.

all_hamilton_cycles_by_DFS(digraf, wazony, sortuj_luki, sortuj_cykle) – wyznacza wszystkie skierowane, ważone lub nie, cykle Hamiltona w grafie skierowanym. Zmienna boolowska wazony określa, czy digraf jest ważony czy nie. Argument sortuj_luki definiuje sposób sortowania wierzchołków na listach sąsiedztw: dowolna liczba >0 (<0) sortuje wierzchołki rosnąco (malejąco) ze względu na wagi łuków wchodzących do wierzchołków na liście sąsiedztwa rozważanego wierzchołka. Wartość 0 sortuje każdą listę sąsiedztw rosnąco ze względu na numer wierzchołka. Jeżeli argument sortuj_cykle jest dowolną liczbą >0 (<0), wtedy cykle Hamiltona zostaną posortowane rosnąco (malejąco) ze względu na ich wagi. Ta funkcja generuje zawsze wszystkie cykle Hamiltona w digrafie, tzn. nie można jej przerwać po wygenerowaniu zadanej liczby cykli. Funkcja działa w oparciu o rekurencyjną procedurę przeszukiwania grafu w głąb (Depth First Search).

all_hamilton_paths(digraf, wazony, v_start, [v_stop]) – wyznacza wszystkie ścieżki Hamiltona w grafie skierowanym ważonym lub nie. Jeżeli drugi argument (wazony) jest true, wtedy digraf jest traktowany jako ważony. Ścieżki Hamiltona będą zaczynać się w wierzchołku v_start. Jeżeli podano czwarty argument, wtedy ścieżki będą kończyć się w podanym wierzchołku.

all_paths(digraf, wazony, v_start, v_stop, [dlugosc]) – wyznacza wszystkie ścieżki w digrafie ważonym lub nie, o dowolnej (>=1) długości, liczonej liczbą łuków w ścieżce,

zaczynające się w wierzchołku v_start i kończące się w wierzchołku v_stop, który może być równy v_start. Jeżeli poda się piąty argument, liczbę całkowitą, wtedy zostaną wyznaczone tylko te ścieżki, które posiadają wymaganą długość.

all_euler_cycles(graf) – wyznacza wszystkie cykle Eulera w grafie nieskierowanym.

```
(%i1056) graf : [[1,2],[2,3],[3,4],[4,1],[3,5],[5,6],[6,7],[7,3]];
         all_euler_cycles(graf);
(graf)   [[1,2],[2,3],[3,4],[4,1],[3,5],[5,6],[6,7],[7,3]]
(%o1056) [[1,2,3,5,6,7,3,4,1],[1,2,3,7,6,5,3,4,1],[1,4,3,5,6,7,3,2,1],
[1,4,3,7,6,5,3,2,1]]
```

random_weighted_graph(n, p, minwaga, makswaga) – generuje losowy graf nieskierowany n-wierzchołkowy, z wagami należącymi do przedziału <minwaga, makswaga>. Prawdopodobieństwo wystąpienia krawędzi wynosi p.

```
(%i1057) random_weighted_graph(5, 0.6, 33, 39);
(%o1057) [[[1,2],37],[[1,4],35],[[1,5],38],[[2,3],38],[[2,4],33],[[
2,5],33],[[3,4],36],[[3,5],36]]
```

random_weighted_digraph(n, p, minwaga, makswaga) – generuje losowy graf skierowany n-wierzchołkowy, z wagami należącymi do przedziału <minwaga, makswaga>. Prawdopodobieństwo wystąpienia łuku wynosi p.

random_Graph(n, p) – generuje losowy n-wierzchołkowy graf nieskierowany bez wag. Prawdopodobieństwo wystąpienia krawędzi wynosi p.

```
(%i1064) random_Graph(5, 0.6);
(%o1064) [[1,2],[1,3],[1,4],[1,5],[2,3],[3,4],[3,5]]
```

random_Digraph(n, p) – generuje losowy n-wierzchołkowy graf skierowany bez wag. Prawdopodobieństwo wystąpienia łuku wynosi p.

tree_from_prufer_code(kod_Prufera, v_start, czy_dac_wagi) – wyznacza krawędzie drzewa nieskierowanego, zakodowanego podanym kodem Prufera. V_start oznacza najmniejszy numer wierzchołka, jaki ma wystąpić w drzewie. Zmienna boolowska czy_dac_wagi decyduje, czy krawędziom drzewa przypisać wagi w postaci kolejnych liczb naturalnych (w kolejności odtwarzania krawędzi z kodu Prufera). Te wagi mogą się przydać podczas

rysowania grafu drzewa. Wtedy wagi pokazują kolejność dodania krawędzi do drzewa.

```
(%i1066) tree_from_prufer_code([3,5,3,3,1], 1, true);
(%o1066) [[[3,2],1],[[5,4],2],[[3,5],3],[[3,6],4],[[1,3],5],[[1,7],6]]
```

prufer_code_for_tree(drzewo, [v_start]) – zwraca kod Prufera dla drzewa nieskierowanego bez wag. Opcjonalna zmienna v_start definiuje minimalny numer wierzchołka, który wystąpi w drzewie. Standardowo jest to 1.

```
(%i1075) drzewo : tree_from_prufer_code([3,5,3,3,1], 1, false);
         kodP : prufer_code_for_tree(drzewo, 1);
(drzewo) [[3,2],[5,4],[3,5],[3,6],[1,3],[1,7]]
(kodP)   [3,5,3,3,1]
```

is_connected_graph(graf) – zwraca true, jeżeli graf nieskierowany jest spójny oraz false w przeciwnym wypadku.

```
(%i1068) graf : [[2,3],[1,4],[5,1],[5,4]];
         is_connected_graph(graf);
(graf)   [[2,3],[1,4],[5,1],[5,4]]
(%o1068) false

(%i1070) graf : [[2,3],[1,4],[5,1],[5,4],[3,5]];
         is_connected_graph(graf);
(graf)   [[2,3],[1,4],[5,1],[5,4],[3,5]]
(%o1070) true
```

is_undirected_tree(graf) – zwraca true, jeżeli graf jest drzewem nieskierowanym oraz false w przeciwnym wypadku.

```
(%i1094) is_undirected_tree([[1,2],[2,3],[4,3],[2,6],[5,6]]);
(%o1094) true

(%i1095) is_undirected_tree([[1,2],[2,3],[4,3],[2,6],[5,6],[1,5]]);
(%o1095) false
```

all_spanning_ditrees(digraf, [vroot]) – wyznacza wszystkie skierowane ukorzenione drzewa rozpinające, zstępujące do wierzchołka o największym numerze (n), tzn. takie, że istnieje skierowana ścieżka od każdego wierzchołka do wierzchołka n. Graf musi być listą łuków, a więc jeżeli interesują nas drzewa w grafie nieskierowanym, to każda jego krawędź musi być rozpisana na dwa łuki (można użyć do tej konwersji funkcji graph_to_digraph()). Jeżeli chcemy, aby korzeń znajdował się w innym wierzchołku niż standardowo (n), wówczas należy

go podać jako drugi argument tej funkcji. Jeżeli wierzchołek korzenia podamy z minusem, wtedy zostaną wyznaczone wszystkie drzewa wstępujące z korzenia, a jeżeli z plusem – zstępujące do niego.

```
(%i1091)  graf : complete_digraph(4);
          drzewa : all_spanning_ditrees(graf,-2);
(graf)    [[1,2],[1,3],[1,4],[2,1],[2,3],[2,4],[3,1],[3,2],[3,4],[4,1]
,[4,2],[4,3]]
(drzewa)  [[[2,1],[1,4],[1,3]],[[2,1],[1,4],[2,3]],[[2,1],[1,4],[4,3]
],[[2,1],[2,4],[1,3]],[[2,1],[2,4],[2,3]],[[2,1],[2,4],[4,3]],[[2,1],
[3,4],[1,3]],[[2,1],[3,4],[2,3]],[[3,1],[1,4],[2,3]],[[3,1],[2,4],[2,
3]],[[3,1],[2,4],[4,3]],[[3,1],[3,4],[2,3]],[[4,1],[2,4],[1,3]],[[4,
1],[2,4],[2,3]],[[4,1],[2,4],[4,3]],[[4,1],[3,4],[2,3]]]
```

graph_to_digraph(graf) – konwertuje każdą krawędź grafu ważonego lub nie, na dwa przeciwnie skierowane łuki digrafu. Przykład:

```
(%i6)     graph_to_digraph([ [2,3], [1,4], [3,1], [2,4] ]);
(%o6)     [[2,3],[3,2],[1,4],[4,1],[3,1],[1,3],[2,4],[4,2]]

(%i4)     graph_to_digraph([ [[2,3],7], [[1,4],12], [[3,1],5], [[2,4],10]]);
(%o4)     [[[2,3],7],[[3,2],7],[[1,4],12],[[4,1],12],[[3,1],5],[[1,3],5],
[[2,4],10],[[4,2],10]]
```

print_list(lista) – wypisuje każdy element listy w nowej linii. Funkcja powstała w celu przejrzystego wypisywania list.

paths_in_rect_grid(m, n, odcinki, warunek) – funkcja w kracie prostokątnej o wymiarach m odcinków jednostkowych w poziomie i n odcinków jednostkowych w pionie wyznacza wszystkie najkrótsze ścieżki (o długości m+n odcinków jednostkowych), prowadzące z lewego dolnego rogu kraty (z punktu (0,0)) do prawego górnego rogu kraty (do punktu (m,n)). Lista *odcinki* zawiera dowolną liczbę odcinków jednostkowych, które są wyróżnione w tej kracie (np. będą zabronione). Warunek może być dowolnym warunkiem logicznym, jaki muszą spełniać odcinki jednostkowe wymienione na liście *odcinki*. Jeżeli wartość logiczna warunku jest true, wtedy ścieżka jest dopisywana do listy wynikowej tej funkcji. Każda ścieżka jest kodowana ciągiem binarnym złożonym z m jedynek i n zer (odcinek poziomy jest kodowany jedynką, a pionowy – zerem). Każdy odcinek definiuje się parą list, podając współrzędne jego początku i końca: [[x1,y1],[x2,y2]].

Przykłady:

Ścieżki, które nie przechodzą przez żaden z trzech wymienionych odcinków

```
(%i179)  odcinki: [ [[3,1],[2,1]], [[4,2],[5,2]], [[1,1],[1,0]] ]$
         warunek(z) :=  not (z[1] or z[2] or z[3])$
         sciezki : paths_in_rect_grid(7,5, odcinki, warunek)$ length(sciezki);
(%o179)  372

(%i191)  odcinki: [ [[3,1],[2,1]], [[4,2],[5,2]], [[1,1],[1,0]] ]$
         warunek(z) :=  not z[1] and not z[2] and not z[3]$
         sciezki : paths_in_rect_grid(7,5, odcinki, warunek)$ length(sciezki);
(%o191)  372
```

Ścieżki, które przechodzą przez odcinek pierwszy albo drugi, albo trzeci, czyli przez tylko jeden z nich:

```
(%i183)  odcinki: [ [[3,1],[2,1]], [[4,2],[5,2]], [[1,1],[1,0]] ]$
         warunek(z) :=  (z[1] and not z[2] and not z[3])
                     or (not z[1] and z[2] and not z[3])
                     or (not z[1] and not z[2] and z[3])$
         sciezki : paths_in_rect_grid(7,5, odcinki, warunek)$ length(sciezki);
(%o183)  290

(%i155)  odcinki: [ [[3,1],[2,1]], [[4,2],[5,2]], [[1,1],[1,0]] ]$
         warunek(z) :=  (z[1] xor z[2] xor z[3]) and not (z[1] and z[2] and z[3])$
         sciezki : paths_in_rect_grid(7,5, odcinki, warunek)$ length(sciezki);
(%o155)  290
```

Ścieżki, które przechodzą przez odcinek pierwszy albo drugi, albo trzeci lub przechodzą przez wszystkie trzy odcinki równocześnie (dlaczego?):

```
(%i187)  odcinki: [ [[3,1],[2,1]], [[4,2],[5,2]], [[1,1],[1,0]] ]$
         warunek(z) :=  z[1] xor z[2] xor z[3]$
         sciezki : paths_in_rect_grid(7,5, odcinki, warunek)$ length(sciezki);
(%o187)  310
```

tsp_for_bakery_and_shops(graf_wazony, sklepy, sortuj_luki, ile_cykli, sciezki_cyklu, [szczegoly]) – funkcja rozwiązuje problem komiwojażera dla jednej piekarni i listy sklepów metodą redukcji wierzchołków digrafu (tzn. metodą stosowaną w kolokwium D). Redukuje się wszystkie wierzchołki z wyjątkiem piekarni i sklepów. W rzeczywistości funkcja nie redukuje wierzchołków, ale wykonuje odpowiednie obliczenia na macierzy kosztów digrafu, co jest wygodniejsze do zaprogramowania. Efekt końcowy jest taki sam, jaki daje redukcja, która jest z kolei

wygodniejsza do przeprowadzenia ręcznego przez człowieka. Znaczenie argumentów sortuj_luki oraz ile_cykli jest takie samo jak dla funkcji all_hamilton_cycles_by_transversals(). Jeżeli argument logiczny sciezki_cyklu jest równy true, wtedy dla każdego cyklu Hamiltona zostanie podana dokładna ścieżka „od wierzchołka do wierzchołka", wraz z wagą całego cyklu. Jeżeli argument sciezki_cyklu jest równy false, wtedy dla każdego cyklu Hamiltona podane będą kolejne łuki z wagami cząstkowymi oraz waga całego cyklu. Jeżeli wywołać tę funkcję z sześcioma argumentami (szósty dowolny), wówczas zobaczymy szczegóły działania tej funkcji tak, że będzie można sprawdzić, czy samodzielnie przeprowadzone obliczenia ręczne są prawidłowe. Pierwsza liczba na liście sklepy oznacza piekarnię. Znalezione cykle startują z piekarni i odwiedzają każdy sklep dokładnie raz, ale niekoniecznie w takiej kolejności, jaka została ustalona na liście sklepy. Od bieżącego sklepu do następnego zostanie znaleziona najtańsza skierowana ścieżka ważona. Będzie ona przechodzić przez jeden lub więcej wierzchołków, które nie są sklepami. Z tego powodu wierzchołek, który nie jest sklepem, może w znalezionym cyklu komiwojażera wystąpić więcej niż raz. Nie jest to błędem, ale wynika to po prostu z przyjętej definicji problemu piekarni i sklepów, gdyż nie jest to klasyczny problem komiwojażera. W problemie (jednej) piekarni i (wielu) sklepów nie odwiedzamy każdego wierzchołka digrafu dokładnie raz, ale tylko wybrane wierzchołki (sklepy) w taki sposób, aby za pośrednictwem wierzchołków-nie sklepów zminimalizować długość cyklu Hamiltona (odwiedzającego wszystkie sklepy).

digraph_to_matrix(digraf, pusta_wartosc) – konwertuje ważony lub nie ważony digraf do (kwadratowej) macierzy kosztów. Pusta_wartosc (zazwyczaj zero lub inf) to wartość, która zostanie wpisana do macierzy kosztów w tych komórkach, dla których nie

istnieją łuki w digrafie. Jeżeli digraf nie jest ważony, wówczas wagi łuków będą przyjęte równe 1.

warshall_shortest_path_algorithm(graf_wazony, pokaz, [kolejnosc]) – funkcja, dla ważonego digrafu, wyznaczy najkrótsze ścieżki oraz odpowiadające im wagi dla wszystkich par wierzchołków w tym digrafie. Oczywiście, ścieżka od i do j to nie to samo, co ścieżka od j do i. Zastosowany zostanie algorytm Warshalla, działający na macierzy kosztów, która jest przekształcana tyle razy, ile jest wierzchołków w digrafie. Algorytmowi temu odpowiada algorytm redukcji wierzchołków digrafu. Jeżeli argument pokaz będzie równy true, wtedy pokazane zostaną kolejne etapy obliczeń na macierzy kosztów. Opcjonalny trzeci argument może być dowolną permutacją zbioru wierzchołków (de facto numerów kolumn macierzy) zapisaną w postaci zwykłej listy, czyli w postaci 1-wierszowej. Można dzięki temu przekonać się, że kolejność „redukowanych" wierzchołków digrafu nie ma wpływu na ostateczny wynik.

dijkstra_shortest_path_algorithm(graf_wazony, vstart, [pokaz]) – funkcja, dla ważonego digrafu, znajduje najkrótsze ścieżki i ich wagi, od ustalonego wierzchołka vstart do wszystkich pozostałych wierzchołków. Algorytm Dijkstry jest więc podobny do algorytmu Warshala, tzn. robi to samo co ten drugi, ale tylko dla jednego wierzchołka. Definicje i złożoności czasowe tych algorytmów są jednak różne. Jeżeli wywołać tę funkcję z trzema argumentami (trzeci dowolny), wtedy można prześledzić działanie kolejnych kroków algorytmu Dijkstry.

grid_Graph(m, n, [waga]) – jeżeli ta funkcja jest wywołana tylko z dwoma argumentami m i n, wtedy zwraca nie ważony graf siatkowy o m wierzchołkach „w poziomie" i n wierzchołkach „w pionie". Jeżeli użyć dwóch dodatkowych argumentów (min_waga i maks_waga), wtedy zwrócony zostanie graf ważony, którego wagi będą należeć do przedziału <min_waga, maks_waga).

octahedral_graph() – zwraca nie ważony graf nieskierowany, utworzony z 6 wierzchołków i 12 krawędzi ośmiościanu foremnego.

icosahedral_graph() – zwraca nie ważony graf nieskierowany, utworzony z 12 wierzchołków i 30 krawędzi dwudziestościanu foremnego.

dodecahedral_graph () – zwraca nie ważony graf nieskierowany, utworzony z 20 wierzchołków i 30 krawędzi dwunastościanu foremnego.

cartesian_prod(zbiory,[nr]) – dla podanego ciągu zbiorów (zapisanego w postaci listy list) funkcja wyznacza iloczyn kartezjański tych zbiorów, czyli generuje wszystkie ciągi na zasadzie „każdy z każdym”. Każdy ciąg składa się z tylu elementów, ile jest zbiorów do pomnożenia. Na i-tej pozycji każdego ciągu znajduje się jakiś element z i-tego zbioru. Jeżeli użyć drugiego opcjonalnego argumentu (i będzie to liczba całkowita), wówczas zostanie wyznaczony ciąg iloczynu kartezjańskiego o podanym numerze. Jeżeli natomiast ten opcjonalny argument będzie jednym z ciągów iloczynu kartezjańskiego tych zbiorów, wtedy zostanie wyznaczony numer tego ciągu. Liczba ciągów iloczynu kartezjańskiego zbiorów jest równa iloczynowi liczb elementów w mnożonych zbiorach.

Pierwotna wersja biblioteki zawierała tylko funkcje działające na permutacjach. Z czasem, w sposób naturalny, z powodu bliskich związków permutacji (kombinatoryki) z grafami, zostały dodane funkcje operujące na grafach, ale nie tylko takie funkcje.

Permutacje w programie Maxima, dzięki funkcjom biblioteki kombinatoryczno-grafowej, można zapisywać na dwa podstawowe sposoby:

- w postaci 1-wierszowej, np.: `[5,1,3,2,4], [3,1,2], [1,2,3,4], [2,4,6,1,3,5],`
- w postaci cyklowej, np.: `[[1],[2],[3],[4]], [[5,2,4],[3,1]], [[6],[4,2]], [[3,5,2,1]], [[2],[1,3],[5]]`. Możliwy jest też zapis macierzowy.

W zapisie 1-wierszowym muszą wystąpić wszystkie liczby od 1 do wybranego n. Największa liczba w permutacji zapisanej 1-wierszowo określa, ile wynosi n.

W zapisie cyklowym można pominąć wszystkie albo niektóre punkty stałe. Największa liczba w permutacji zapisanej cyklowo określa, ile wynosi n. Jeżeli więc np. liczba 6 jest punktem stałym pewnej permutacji i ta permutacja ma należeć do S_6, to tego punktu stałego

nie można pominąć! Na przykład, permutacja `[[2,5], [1,4,3]]` zostanie rozpoznana jako należąca do S_5, natomiast permutacja `[[2,5], [6], [1,4,3]]` - jako należąca do S_6.

Aby swobodnie korzystać z funkcji biblioteki, należy nauczyć się zasad prowadzenia obliczeń w środowisku Maxima. Na stronie internetowej tego programu są różne tutoriale, które wyjaśniają, jak zacząć. Tutaj zostaną podane tylko bardzo podstawowe porady.

Operatorem przypisania w programie Maxima jest dwukropek, np.:

```
p1:[[6,2,3],[1]];  p2:[5,2,1,4,3];
```

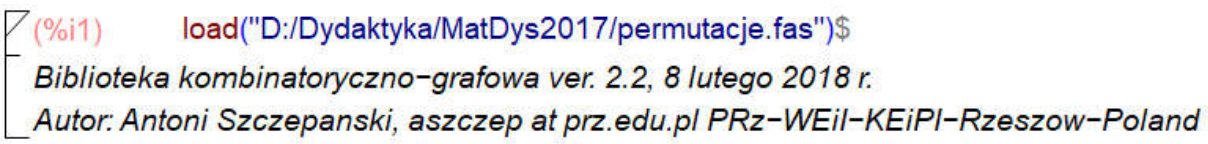
Funkcje biblioteki, dotyczące permutacji, można wywoływać na permutacjach zapisanych w zmiennych lub podanych bezpośrednio w postaci 1-wierszowej lub cyklowej. Można te dwa podejścia dowolnie mieszać. Większość funkcji, dla których argumentem jest permutacja, akceptuje permutacje podane w obu zapisach. Niektóre funkcje, jeżeli nie jest to sprzeczne z ich działaniem, zwracają na wyjściu taki zapis permutacji, jaki otrzymały na wejściu.

Komórkę w Maximie oblicza się wciskając równocześnie klawisze Shift i Enter. Każdą instrukcję należy kończyć średnikiem albo symbolem $. Symbol $ stosuje się wówczas, gdy użytkownik nie chce, aby wyniki obliczeń zostały wyświetlone (np. z powodu ich dużej objętości tekstowej). W jednej komórce można wprowadzić wiele instrukcji, przechodząc do następnej linii za pomocą klawisza Enter.

Aby załadować bibliotekę kombinatoryczno-grafową do środowiska obliczeniowego Maximy, należy w programie Maxima wpisać instrukcję ładowania pakietu z określonej lokalizacji na dysku komputera, np.:

```
load("C:\\Studia\\MatDys\\Permutacje\\permutacje.fas");   lub
load("C:/Studia/MatDys/Permutacje/permutacje.fas");
```

i nacisnąć kombinację klawiszy Shift+Enter. Taki sam efekt można uzyskać w inny sposób. Ten drugi sposób załadowania biblioteki jest prostszy, ponieważ nie wymaga wpisywania z klawiatury ścieżki prowadzącej do pliku o nazwie permutacje.fas. Z menu Plik okienka Maximy należy wybrać opcję "Wczytaj pakiet". W oknie wyszukiwania plików, które się otworzy, należy przejść do folderu, w którym znajduje się plik permutacje.fas i (po zmianie filtra plików z "Pakiet Maxima (*.mac)" na "Wszystkie") wybrać ten plik (klikając lewym klawiszem myszy). Jeżeli w wyniku opisanej operacji, po kilku sekundach, pojawi się napis w stylu:



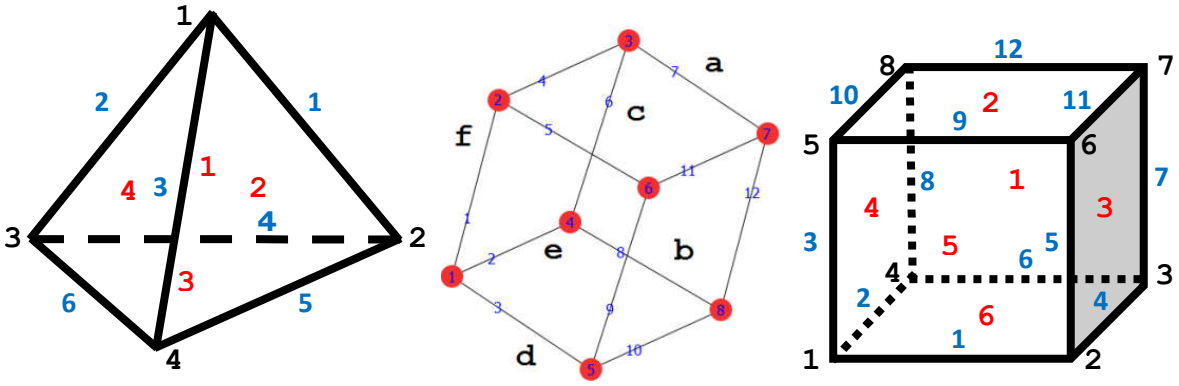
to znaczy, że biblioteka została załadowana pomyślnie.

Biblioteka permutacyjno-grafowa powinna działać w wersji 5.41 i wyższej programu Maxima, zainstalowanego z pliku maxima-clisp-sbcl-5.41.0a-win64.exe, pobranego ze strony https://sourceforge.net/projects/maxima/files/Maxima-Windows/5.41.0-Windows/

Po załadowaniu biblioteki kombinatoryczno-grafowej użytkownik ma do dyspozycji nie tylko funkcje tej biblioteki, ale także wszystkie inne wbudowane w program Maxima. Aby w pełni wykorzystać możliwości biblioteki funkcji permutacyjnych, należy dobrze znać zasady i możliwości operowania na listach, typie wbudowanym w Maximę. O listach można poczytać w rozdziale 5.4.2. systemu pomocy.

Funkcje działające na grafach wymagają grafu zdefiniowanego jako listy krawędzi lub listy łuków, z wagami lub bez. Nie podaje się listy wierzchołków, więc nie ma możliwości definiowania izolowanych wierzchołków w grafach, chociaż grafy niespójne są dopuszczalne (np. [[1,2], [3,4]]). To ograniczenie być może zostanie kiedyś usunięte. Nazwy niektórych funkcji grafowych zostały zmienione, gdyż były identyczne jak te, w bibliotece wbudowanej w Maximę - load(graphs). Na przykład funkcja path_Graph() ma dużą literę G, ponieważ w bibliotece graphs istnieje funkcja path_graph(). Możliwe, że w kolejnej wersji biblioteki kombinatoryczno-grafowej funkcje kolidujące z funkcjami biblioteki graphs będą miały inaczej zmienione nazwy (np. path__graph()). Obie biblioteki można załadować i używać równocześnie. W szczególności można graf utworzyć funkcją create_graph() lub inną, a następnie funkcją edges() otrzymać listę krawędzi wykorzystywaną w bibliotece kombinatoryczno-grafowej. Można też postąpić odwrotnie, tzn. dla listy krawędzi dorobić ręcznie lub automatycznie listę wierzchołków i funkcją create_graph() wygenerować graf, na którym można działać za pomocą funkcji dostępnych po load(graphs).

Błędy w działaniu zaimplementowanych funkcji lub propozycje dodatkowych funkcji można zgłaszać osobiście lub pisząc na adres e-mail: aszczep@prz.edu.pl. Zgłoszenie błędu najlepiej poprzeć kopią ciągu instrukcji wprowadzonych w komórkach Maximy, które spowodowały błąd.

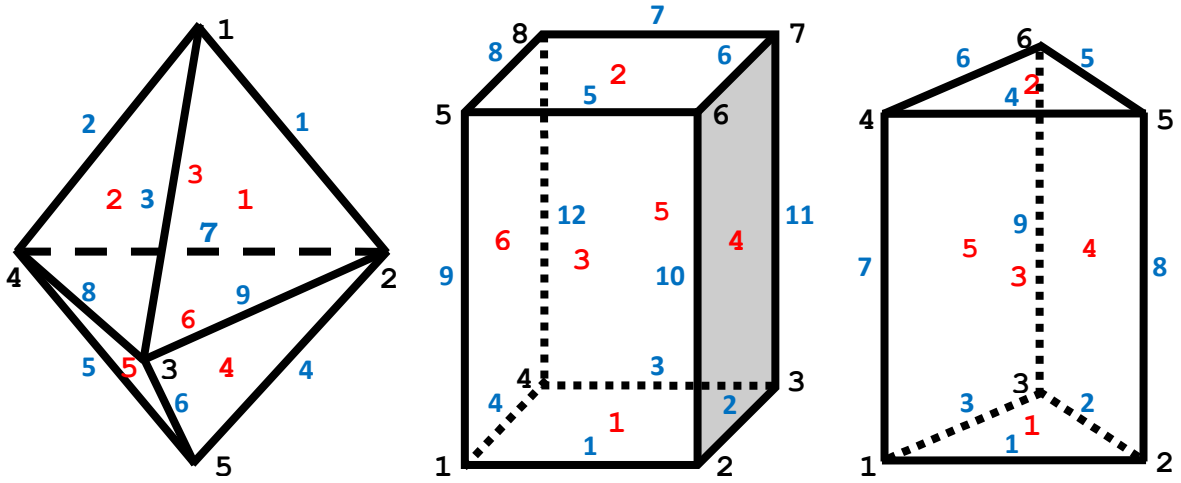


Oznaczenia wierzchołków, krawędzi i ścian czworościanu foremnego oraz sześcianu dla funkcji:
`p_group_tetraedr_vertices()` `p_group_tetraedr_faces()`
`p_group_tetraedr_edges()`

```
p_group_isometries_hexaedr_vertices()
p_group_isometries_hexaedr_edges()
p_group_isometries_hexaedr_faces()
hexaedr_graph()
hexaedr_faces_adjacency_graph()
```

```
p_group_hexaedr_vertices()
p_group_hexaedr_edges()
p_group_hexaedr_faces()
```

Literom a, b, c, d, e, f, oznaczającym ściany sześcianu, odpowiadają liczby 1, 2, 3, 4, 5, 6.



Oznaczenia wierzchołków, krawędzi i ścian bryły powstałej z połączenia dwóch czworościanów foremnych oraz prostopadłościanów o podstawie kwadratowej (prostokątnej) lub trójkątnej, dla funkcji:

```
p_group_two_tetraedrs_vertices()
p_group_two_tetraedrs_edges()
p_group_two_tetraedrs_faces()
```

```
p_group_not_right_cuboid_vertices()
p_group_not_right_cuboid_edges()
p_group_not_right_cuboid_faces()
```

```
p_group_right_cuboid_vertices()
p_group_right_cuboid_edges()
p_group_right_cuboid_faces()
```

```
p_group_triangular_cuboid_vertices()
p_group_triangular_cuboid_edges()
p_group_triangular_cuboid_faces()
```

Opracował Antoni Szczepański, 8 lutego 2018r.