LMD
Práctica 3. Grafos.

En esta práctica estudiaremos con la ayuda de Maxima algunos conceptos sobre grafos. Para ello hemos de cargar el paquete *graphs*.

```
(%ixx) load(graphs)$
```

Para crear un grafo (no dirigido) podemos usar la función `create_graph` que requiere dos argumentos que son listas. La primera es la lista de los vértices y la segunda es la lista de los lados ó flechas. Cada lado, a su vez, es una lista con los vértices que une el lado.

Por ejemplo, la siguiente línea crea un grafo isomorfo al grafo completo K_3, que almacenamos en una variable `K3`.

```
(%ixx) K3:create_graph([1,2,3], [[1,2],[1,3],[2,3]]);
```

La respuesta de Maxima nos dice que hemos definido un grafo (no dirigido) con tres vértices y tres lados. Con el comando `print_graph` obtenemos información sobre dicho grafo. Si introducimos

```
(%ixx) print_graph(K3);
```

Maxima nos devuelve lo siguiente:

```
Graph on 3 vertices with 3 edges
Adjacencies:
  3 :  2  1
  2 :  3  1
  1 :  3  2
(%oxx) done
```

Es decir, `K3` es un grafo con tres vértices y tres lados. El vértice 3 es adyacente a los vértices 2 y 1 (eso nos indica la primera línea), el vértice 2 es adyacente a los vértices 3 y 1, y el vértice 1 es adyacente a los vértices 3 y 2.

Si hemos definido un grafo, podemos preguntarle a Maxima por los vértices y los lados de dicho grafo. Ésto se hace con los comandos `vertices` y `edges`, respectivamente, y que ilustramos con el grafo `K3`.

```
(%ixx) vertices(K3);
       edges(K3);
```

Para que Maxima dibuje un grafo, usamos el comando `draw_graph`.

```
(%ixx) draw_graph(K3);
```

Como se puede apreciar, Maxima no muestra los nombres de los vértices. Para que lo haga, usamos la opción `show_id=true`. Más adelante veremos más opciones del comando `draw_graph`.

```
(%ixx) draw_graph(K3, show_id=true);
```

Si lo que queremos es crear un grafo dirigido, dentro del comando `create_graph` introduciremos la opción `directed=true`.

```
(%ixx) T3:create_graph([1,2,3], [[1,2],[2,3],[3,1]], directed=true);
```

Ahora le pedimos información sobre el grafo dirigido `T3` que acabamos de crear.

```
(%ixx) print_graph(T3);
```

Ésto produce la salida siguiente:

```
Digraph on 3 vertices with 3 arcs
Adjacencies:
  3 :  1
  2 :  3
  1 :  2
(%oxx) done
```

La palabra *Digraph* que muestra Maxima en su salida es una abreviatura del término inglés *Directed graph*. Observe ahora cómo a la hora de definir las adyacencias entre vértices influye el sentido de las flechas. Por ejemplo, hemos definido en `T3` el lado `[3,1]` lo cual significa que hay una flecha del nodo `3` al nodo `1`, pero no al revés. Le pedimos que lo dibuje.

```
(%ixx) draw_graph(T3, show_id=true);
```

También podemos preguntarle por los vértices y los lados del grafo dirigido `T3` que hemos definido.

```
(%ixx) vertices(T3);
       edges(T3);
```

El comando `is_digraph` comprueba si un grafo es dirigido, mientras que el comando `is_graph` comprueba si es no dirigido. Pruebe los siguientes ejemplos:

```
(%ixx) is_graph(K3);
       is_graph(T3);
       is_digraph(K3);
       is_digraph(T3);
```

En la versión actual de Maxima sólo podemos definir grafos, ya sean dirigidos o no dirigidos, en los que no hay autolazos ni lados paralelos. Cualquiera de las tres instrucciones siguientes, da un mensaje de error y no crea ningún grafo.

```
(%ixx) G1:create_graph([1,2,3], [[1,2],[2,3],[3,1],[1,2]]);
(%ixx) G2:create_graph([1,2,3], [[1,2],[2,3],[3,1],[2,1]]);
(%ixx) G3:create_graph([1,2,3], [[1,2],[2,3],[3,1],[1,1]]);
```

En el caso de `G1` y de `G2`, detecta que el lado $[1,2]$ ha sido definido dos veces. Como estamos intentando crear grafos no dirigidos, considera el lado $[1,2]$ igual al lado $[2,1]$.

Sobre G3, nos dice que $[1,1]$ no es un lado válido. Comprobamos que ningún grafo ha sido asignado a las variables G1, G2 o G3.

```
(%ixx) G1; G2; G3;
```

Sin embargo, en el caso de grafos dirigidos, para cada par de vértices distintos u y v, permite definir una flecha de u a v y otra de v a u.

```
(%ixx)  GD:create_graph([1,2,3], [[1,2],[2,1],[2,3]], directed=true);
        draw_graph(GD, show_id=true);
```

En este ejemplo hemos definido una flecha del vértice 1 al vértice 2, y otra del vértice 2 al vértice 1. También hemos definido una flecha del vértice 2 al vértice 3, pero no hemos definido ninguna flecha del vértice 3 al vértice 2.

Tanto para grafos dirigidos como para no dirigidos, es posible asignarle un peso ó cantidad numérica a cada arista. A modo de ejemplo redefinimos el grafo no dirigido K3 y el grafo dirigido T3 asignándoles pesos a sus lados.

```
(%ixx) K3:create_graph([1,2,3], [[[1,2],8],[[2,3],1/2],[[1,3],4.5]]);
       T3:create_graph([1,2,3], [[[1,2],5/3],[[2,3],7],[[1,3],0.4]],
          directed=true);
```

Por ejemplo, en K3 le hemos asignado el peso 8 al lado $[1,2]$, el peso $1/2$ al lado $[2,3]$, y el peso 4.5 al lado $[1,3]$, y similarmente para T3.

```
(%ixx) print_graph(K3);
       print_graph(T3);
```

Observe que los pesos no son mostrados por el comando print_graph. El comando draw_graph sí lo hace, aunque para ello hemos de incluir la opción show_weight=true.

```
(%ixx) draw_graph(K3, show_id=true, show_weight=true);
       draw_graph(T3, show_id=true, show_weight=true);
```

En el paquete *graphs* también hay algunos tipos de grafos que ya vienen predefinidos. Veamos algunos ejemplos:

```
(%ixx) K6:complete_graph(6)$
       draw_graph(K6, show_id=true);
```

Así, primero hemos creado un grafo completo con 6 vértices, el cual lo hemos almacenado en la variable K6 y a continuación lo hemos representado en pantalla.

```
(%ixx) K34:complete_bipartite_graph(4,3)$
       draw_graph(K34, show_id=true);
```

Ahora hemos creado y dibujado un grafo bipartido completo con particiones de 4 y 3 vértices, respectivamente. A continuación obtenemos y dibujamos el complementario del grafo K34, es decir, el grafo que tiene los mismos vértices que K34 y los lados que le faltan a K34 para ser completo.

```
(%ixx) CK34:complement_graph(K34)$
       draw_graph(CK34, show_id=true);
```

Lo más seguro es que el dibujo obtenido no refleje realmente cómo es el grafo `CK34`, pues los vértices en cada componente conexa son representados de forma alineada. Ello provoca que algunos lados no se visualicen correctamente. Para evitar este problema, incluímos la opción `redraw=true` que recalcula las posiciones de los vértices y así obtenemos un dibujo más fidedigno del grafo.

```
(%ixx) draw_graph(CK34, redraw=true);
```

Al margen de estos detalles sobre la representación, ahora le pedimos que nos muestre la información sobre los últimos grafos que hemos definido. Nótese que los comandos de Maxima que generan grafos, numeran los vértices comenzando en 0.

```
(%ixx) print_graph(K6); print_graph(K34); print_graph(CK34);

Graph on 6 vertices with 15 edges
Adjacencies:
  5 :  4  3  2  1  0
  4 :  5  3  2  1  0
  3 :  5  4  2  1  0
  2 :  5  4  3  1  0
  1 :  5  4  3  2  0
  0 :  5  4  3  2  1
(%oxx) done

Graph on 7 vertices with 12 edges
Adjacencies:
  6 :  3  2  1  0
  5 :  3  2  1  0
  4 :  3  2  1  0
  3 :  6  5  4
  2 :  6  5  4
  1 :  6  5  4
  0 :  6  5  4
(%oxx) done

Graph on 7 vertices with 9 edges
Adjacencies:
  0 :  1  2  3
  1 :  0  2  3
  2 :  0  1  3
  3 :  0  1  2
  4 :  5  6
  5 :  4  6
  6 :  4  5
(%oxx) done
```

En el caso de `K34`, que es un grafo bipartido completo, observe que el conjunto de vértices $V = \{0, 1, 2, 3, 4, 5, 6\}$ es particionado por Maxima como $V = V_1 \cup V_2$, donde

$$V_1 = \{4, 5, 6\} \qquad \text{y} \qquad V_2 = \{0, 1, 2, 3\},$$

y los únicos lados permitidos son entre vértices de V_1 y vértices de V_2.
Veamos algunos ejemplos más.

```
(%ixx) draw_graph(cycle_graph(7), show_id=true);
```

El comando anterior dibuja un grafo que es un ciclo de 7 vértices, mientras que el siguiente dibuja un grafo que es un ciclo dirigido de 7 vértices.

```
(%ixx) draw_graph(cycle_digraph(7), show_id=true);
```

A continuación representamos un grafo que es un camino de cinco vértices y otro que es un camino dirigido de cinco vértices.

```
(%ixx) draw_graph(path_graph(5), show_id=true);
```

```
(%ixx) draw_graph(path_digraph(5), show_id=true);
```

También podemos crear un grafo a partir de su matriz de adyacencia. Para ello primero definimos una matriz (simétrica de ceros y unos, con ceros en su diagonal principal) que llamamos `A` y que es la matriz de adyacencia de un grafo. A continuación, le aplicamos el comando `from_adjacency_matrix` que crea el grafo correspondiente a dicha matriz.

```
(%ixx) A:matrix([0,1,1,1],[1,0,1,1],[1,1,0,1],[1,1,1,0])$
       K4:from_adjacency_matrix(A);
```

De esta forma hemos creado un grafo completo de cuatro nodos que llamamos `K4`. Para convencernos de ello, podemos decirle que nos lo dibuje, o podemos preguntarle directamente si dicho grafo es isomorfo a un grafo completo con 4 vértices. Optamos por la segunda opción y para ello ejecutamos el comando siguiente.

```
(%ixx) is_isomorphic(K4,complete_graph(4));
```

Como la respuesta de Maxima es `true`, podemos pedirle que obtenga un isomorfismo concreto del grafo que hemos definido en un grafo completo de 4 vértices. Para ello escribimos lo siguiente.

```
(%ixx) isomorphism(K4,complete_graph(4));
```

Nótese que tanto en el grafo `K4` que hemos definido a partir de su matriz de adyacencia, como en el grafo `complete_graph(4)`, Maxima numera los vértices con los números 0,1,2,3.

Si definimos un grafo y lo asignamos a una variable `G1` y ésta a continuación la asignamos a otra variable `G2`, realmente ambas variables representan el mismo grafo, de modo que si modificamos el grafo representado por una de ellas, también estamos modificando el grafo representado por la otra variable. Veamos un ejemplo de ésto.

```
(%ixx) G1:complete_graph(4);
       G2:G1;
```

```
(%ixx) draw_graph(G1, show_id=true);
       draw_graph(G2, show_id=true);
```

Ahora ejecutamos el comando `remove_edge([1,2], G1)` que elimina el lado `[1,2]` del grafo `G1`, sin eliminar los vértices en los que se sustenta dicho lado.

```
(%ixx) remove_edge([1,2], G1);
```

```
       draw_graph(G1, show_id=true);
       draw_graph(G2, show_id=true);
```

Como se aprecia en los dibujos, el lado `[1,2]` ha sido suprimido en los grafos almacenados en las variables `G1` y `G2`, porque ambas variables apuntan a la misma posición de memoria donde se almacena nuestro grafo.

Para evitar ésto, o definimos cada grafo por separado, o bien usamos el comando `copy_graph` que crea una nueva copia de un grafo ya existente.

Introduzca los siguientes comandos.

```
(%ixx) G1:complete_graph(4);
       G2:copy_graph(G1);

(%ixx) draw_graph(G1, show_id=true);
       draw_graph(G2, show_id=true);

(%ixx) remove_edge([1,2], G1);
       draw_graph(G1, show_id=true);
       draw_graph(G2, show_id=true);
```

Como se aprecia en los dibujos, el lado `[1,2]` ha sido suprimido en el grafo almacenado en la variable `G1`, pero no en el grafo almacenado en la variable `G2`.

Maxima incluye otras construcciones típicas con grafos como son la unión y el producto cartesiano de dos grafos.

Definimos y dibujamos dos grafos camino de 4 y 5 vértices, que almacenamos en las variables `G1` y `G2`, respectivamente.

```
(%ixx) G1:path_graph(5);
       G2:path_graph(4);

(%ixx) draw_graph(G1, show_id=true);
       draw_graph(G2, show_id=true);
```

Ahora definimos y dibujamos un nuevo grafo que es la unión de ambos, y que almacenamos en la variable `G`. Además le preguntamos por los conjuntos de los vértices de cada uno de los grafos.

```
(%ixx) G:graph_union(G1,G2);
       draw_graph(G, show_id=true);

(%ixx) vertices(G1); vertices(G2); vertices(G);
```

Ésto nos indica que Maxima, si es necesario, renombra los vértices al hacer la unión de dos grafos. Ejecute también el siguiente comando el cual devuelve una lista que contiene dos listas. Cada una de estas últimas está formada por los vértices de una componente conexa del grafo `G`.

```
(%ixx) connected_components(G);
```

Introduzca también los siguientes comandos.

```
(%ixx) H:graph_union(G1,G1,G1,G2);
```

```
(%ixx) connected_components(H);

(%ixx) draw_graph(H, show_id=true);
```

Ejercicio. Defina en Maxima el grafo $K_4 \cup \overline{K}_{4,6}$ y escriba un comando para obtener sus componentes conexas. □

A partir de dos grafos G_1 y G_2 podemos construir un nuevo grafo, que representaremos por $G_1 \times G_2$, denominado el *producto cartesiano* de G_1 y G_2. Vea el Ejercicio 50 de la Relación de problemas del Tema 3. A pesar de que la definición dada allí puede resultar en un principio algo extraña, la idea intuitiva subyacente es muy simple. Para ilustrarlo, ejecute algunos de los comandos siguientes.

```
(%ixx) draw_graph(graph_product(path_graph(2),path_graph(3)));

(%ixx) draw_graph(graph_product(path_graph(3),path_graph(5)));

(%ixx) draw_graph(graph_product(path_graph(1),path_graph(3)));

(%ixx) draw_graph(graph_product(cycle_graph(4),path_graph(3)));

(%ixx) draw_graph(graph_product(cycle_graph(4),path_graph(5)));

(%ixx) draw_graph(graph_product(cycle_graph(3),cycle_graph(10)));

(%ixx) draw_graph(graph_product(cycle_graph(4),cycle_graph(10)));
```

El comando `graph_product` puede ser útil para responder a los distintos apartados de dicho ejercicio.

Una vez que tenemos un grafo, podemos preguntarnos cosas sobre él. Recordemos que la variable `K34` almacenaba un grafo bipartido completo $K_{3,4}$.

```
(%ixx) draw_graph(K34, show_id=true);
```

El grado de un vértice `v` en un grafo `G` se obtiene con `vertex_degree(v,G)`. Con los comandos siguientes, le preguntamos el grado de los vértices 1 y 6 en `K34`, respectivamente.

```
(%ixx) vertex_degree(1,K34);
       vertex_degree(6,K34);
```

Una secuencia de grados para `K34` podemos obtenerla con

```
(%ixx) degree_sequence(K34);
```

Recordemos en este sentido que en Maxima es posible ordenar listas numéricas de forma creciente y decreciente con los comandos `sort` y `reverse`, respectivamente.

Comprobamos a continuación el *Teorema de Euler de los grados* para el grafo `K34`. La suma de los grados de los vértices de `K34` podemos obtenerla con

```
(%ixx) apply("+",degree_sequence(K34));
```

Lo que hacemos es aplicar el operador suma a los elementos de la lista de grados, con lo que obtenemos la suma de los grados de los vértices de `K34`. Ya que los vértices del grafo son 0,1,2,3,4,5,6, otra forma de lograr ésto es la siguiente:

```
(%ixx) sum(vertex_degree(v,K34),v,0,6);
```

Por otra parte, `edges(K34)` es la lista de los lados del grafo `K34` y `length(edges(K34))` es la longitud de la misma, es decir, el número de lados en `K34`. Por tanto, constatamos el Teorema de Euler para el grafo `K34` mediante cualquiera de los dos comandos siguientes:

```
(%ixx) apply("+",degree_sequence(K34)) = 2*length(edges(K34));
```

```
(%ixx) is(apply("+",degree_sequence(K34)) = 2*length(edges(K34)));
```

Si `G` es un grafo dirigido, el grado de entrada y el grado de salida de un vértice `v` en `G` se obtiene con `vertex_in_degree(v,G)` y `vertex_out_degree(v,G)`, respectivamente.

La variable `T3` almacenaba un grafo ciclo dirigido de tres vértices.

```
(%ixx) draw_graph(T3, show_id=true);
```

Obtenemos los grados de entrada y de salida del vértice 2 en `T3`.

```
(%ixx) vertex_in_degree(2,T3);
       vertex_out_degree(2,T3);
```

El comando `random_graph(n, p)` genera aleatoriamente un grafo de `n` vértices, donde la probabilidad de que cada lado esté presente en el grafo es igual a `p`. Pruebe los siguientes comandos.

```
(%ixx) G1:random_graph(10, 1/2);
       G2:random_graph(10, 1/2);
       G3:random_graph(10, 1/2);
```

```
(%ixx) draw_graph(G1, show_id=true);
       draw_graph(G2, show_id=true);
       draw_graph(G3, show_id=true);
```

Veamos si algunos de los grafos generados son isomorfos.

```
(%ixx) is_isomorphic(G1,G2);
       is_isomorphic(G1,G3);
       is_isomorphic(G2,G3);
```

La salida de los comandos siguientes podemos imaginarla sin necesidad de ejecutar tales comandos. Mire las probabilidades que estamos especificando.

```
(%ixx) G1:random_graph(10, 1);
       G2:random_graph(10, 0);
```

```
(%ixx) degree_sequence(G1);
       degree_sequence(G2);
```

```
(%ixx) draw_graph(G1, show_id=true);
       draw_graph(G2, show_id=true);
```

Pruebe también los siguientes comandos, los cuales generan de forma aleatoria árboles de 12 vértices y los dibujan.

```
(%ixx) T1:random_tree(12);
       T2:random_tree(12);
```

```
(%ixx) draw_graph(T1, show_id=true);
```

```
draw_graph(T2, show_id=true);
```

Constatamos que ambos son árboles y comprobamos si son isomorfos.

```
(%ixx) is_tree(T1);
       is_tree(T2);
```

```
(%ixx) is_isomorphic(T1, T2);
```

En la ayuda de Maxima puede encontrar los comandos `random_bipartite_graph`, `random_regular_graph` y `random_digraph` que generan aleatoriamente otros tipos de grafos.

El comando `adjacency_matrix(G)` devuelve la matriz de adyacencia de un grafo `G`. Obtenemos a modo de ejemplo la matriz de adyacencia de nuestro grafo `K6` y la almacenamos en la variable `B`.

```
(%ixx) B:adjacency_matrix(K6);
```

Por el *Teorema del número de caminos*, para cualquier número natural r, los elementos de la matriz $\mathtt{B}^r$ nos indican el número de caminos de longitud r que hay entre los vértices del grafo. Calculamos la quinta potencia de la matriz `B`.

```
(%ixx) B^^5;
```

En vista del resultado obtenido, desde un vértice cualquiera a sí mimo hay 520 lazos de longitud 5, y entre dos vértices distintos hay 521 caminos de longitud 5.

Ejercicio. Calcule el número de caminos de longitud 15 entre dos vértices adyacentes cualesquiera del grafo ciclo C_7. □

Ejercicio. En el grafo completo K_5, cuyos vértices etiquetamos como 0,1,2,3,4, calcule el número de caminos de longitud 11 entre los vértices 3 y 1. ¿Cuántos de tales caminos pasan al menos una vez por el vértice 0? □

Ejercicio. Sea el grafo dirigido G, con $\mathrm{V}(G) = \{v_1, v_2, v_3, v_4, v_5\}$, dado por la matriz de adyacencia siguiente:

$$A = \begin{pmatrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 3 \\ 1 & 0 & 2 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

Obtenga el número de caminos de longitud 16 desde el vértice v_1 al vértice v_2. ¿Cuántos de tales caminos nunca pasan por el vértice v_5? □

Maxima no incorpora un comando específico para comprobar si un grafo `G` tiene o no un camino de Euler. Sin embargo, podemos comprobar ésto fácilmente viendo primero si `G` es conexo y a continuación inspeccionando la secuencia de grados de `G`.

```
(%ixx) G:complete_bipartite_graph(4,6);
```

```
(%ixx) is_connected(G);
       degree_sequence(G);
```

Por tanto el grafo dado contiene lazos de Euler.

Podemos preguntarle a Maxima si en un grafo hay o no un ciclo de Hamilton con el comando `hamilton_cycle`. Caso de existir, nos devuelve una lista con los vértices que forman el ciclo de Hamilton, o bien la lista vacía si no existiera tal ciclo. Recordemos que la variable `K6` almacenaba un grafo completo con seis vértices.

```
(%ixx) hc:hamilton_cycle(K6);
```

Tenemos la posibilidad de que nos dibuje el grafo, y nos marque este ciclo. Para eso, hemos de decirle que nos muestre los lados que unen los vértices del ciclo.

```
(%ixx) draw_graph(K6,show_id=true, show_edges=vertices_to_cycle(hc));
```

Veamos otro ejemplo más interesante. Introduzca los comandos siguientes.

```
(%ixx) G:graph_product(path_graph(4), cycle_graph(5));
       hc:hamilton_cycle(G);
```

```
(%ixx) draw_graph(G,show_id=true, show_edges=vertices_to_cycle(hc));
```

Hemos creado un grafo G que es el producto cartesiano de los grafos P_4 y C_5. A continuación Maxima ha encontrado el ciclo de Hamilton

$$[19, 15, 11, 7, 3, 2, 6, 10, 14, 13, 9, 5, 1, 0, 4, 8, 12, 16, 17, 18, 19],$$

y lo ha representado sobre el grafo.

Cuando no hay ciclo de Hamilton, devuelve la lista vacía, como ocurre en el ejemplo siguiente.

```
(%ixx) G:graph_product(path_graph(5), path_graph(5));
       hc:hamilton_cycle(G);
```

Ejercicio. Utilice el comando `dodecahedron_graph()` para obtener el grafo denominado *dodecaedro de Hamilton* y a continuación escriba instrucciones para encontrar y dibujar un ciclo de Hamilton en dicho grafo. □

También tenemos disponible un comando para buscar un camino abierto de Hamilton. Sabemos que cuando existe un ciclo de Hamilton, también hay un camino abierto de Hamilton. Constamos ésto para uno de los ejemplos anteriores. Introduzca las intrucciones siguientes.

```
(%ixx) G:graph_product(path_graph(4), cycle_graph(5));
       hp:hamilton_path(G);
```

```
(%ixx) draw_graph(G, show_id=true, show_edges=vertices_to_path(hp));
```

Obtenemos la lista siguiente:

$$[15, 11, 7, 3, 2, 6, 10, 14, 13, 9, 5, 1, 0, 4, 8, 12, 16, 17, 18, 19].$$

Puede ocurrir que exista camino abierto de Hamilton, pero no haya ciclo de Hamilton. Ilustramos ésto con el grafo siguiente, para el cual ya hemos constatado más arriba que no contiene ningún ciclo de Hamilton.

```
(%ixx) G:graph_product(path_graph(5), path_graph(5));
       hp:hamilton_path(G);
```

```
(%ixx) draw_graph(G, show_id=true, show_edges=vertices_to_path(hp));
```

Los comandos `hamilton_cycle` y `hamilton_path` son sólo para grafos no dirigidos. Si intentamos aplicarlos a grafos dirigidos, devuelven un mensaje de error. Recordemos nuestro grafo dirigido `T3`.

```
(%ixx) draw_graph(T3, show_id=true);

(%ixx) hc:hamilton_path(T3);

(%ixx) hp:hamilton_path(T3);
```

Para obtener una geodésica ó camino más corto entre dos vértices `u` y `v` de un grafo `G`, tenemos el comando `shortest_path(u,v,G)`. Si `u` y `v` están en componentes conexas distintas, devuelve la lista vacía.

```
(%ixx) G:create_graph([1,2,3,4,5],[[1,2],[2,3],[3,4]]);
       draw_graph(G, show_id=true);

(%ixx) shortest_path(1,4,G);
       shortest_path(1,5,G);
```

Veamos otro ejemplo.

```
(%ixx) G:graph_product(path_graph(4),cycle_graph(5));
       geodesica:shortest_path(3,12,G);

(%ixx) draw_graph(G,show_id=true,
                  show_edges=vertices_to_path(geodesica));
```

Obtenemos la geodésica $[3, 19, 15, 14, 13, 12]$.

Si los lados del grafo tienen asignadas longitudes ó pesos, es decir, números reales no negativos, el comando `shortest_weighted_path(u, v, G)` devuelve una geodésica desde `u` hasta `v`, así como su longitud. Si una arista no tiene peso asignado, su valor por defecto es 1.

```
(%ixx) K10:complete_graph(10)$
```

En la variable `K10` hemos almacenado un grafo completo de 10 vértices. Ahora le asignamos aleatoriamente a cada lado un número entre 0 y 1.

```
(%ixx) for e in edges(K10) do set_edge_weight(e, random(1.0), K10)$
```

A continuación buscamos un camino en `K10` desde el vértice 0 al 9 para el cual la suma de los pesos asignados sea mínima.

```
(%ixx) camino:shortest_weighted_path(0, 9, K10);

(%ixx) draw_graph(K10, show_id=true,
                  show_edges=vertices_to_path(camino[2]));
```

Ejercicio. Escriba algunas instrucciones en Maxima para implementar el Algoritmo de Floyd. Para ello, suponemos que tenemos ya definida la matriz C de costos de orden $n \times n$ donde el valor ∞ lo representamos como una cantidad numérica mayor que la suma de los pesos asignados a todas las flechas del grafo. Puede usar instucciones como las siguientes:

- `zeromatrix`, que devuelve una matriz nula de un tamaño dado.

- `for k:1 thru n do`, que establece un bucle o iteración para la variable `k` desde el valor `1` hasta `n`. Además varios bucles pueden anidarse.
- `if` condición `then` instrucciones, que establece un condicional.
- `block(` instrucciones `)`, que permite agrupar varias instrucciones en un bloque. Tales instrucciones irán separadas por comas. □

Ya hemos usado en un ejemplo previo el comando `is_connected` que permite saber si un grafo (no dirigido) es conexo. Si aplicamos este comando a un grafo dirigido, se obtiene un error.

```
(%ixx) draw_graph(T3, show_id=true);
       is_connected(T3);
```

La versión actual de Maxima no incorpora ningún comando directo para saber si un grafo dirigido es conexo. Sin embargo, con el comando `is_sconnected` podemos saber si un grafo dirigido es fuertemente conexo.

```
(%ixx) GD1:create_graph([1,2,3], [[1,2],[2,3],[3,1]], directed=true);
       GD2:create_graph([1,2,3], [[1,2],[2,3],[1,3]], directed=true);
       GD3:create_graph([1,2,3], [[1,2],[2,1]], directed=true);

(%ixx) draw_graph(GD1, show_id=true);
       draw_graph(GD2, show_id=true);
       draw_graph(GD3, show_id=true);

(%ixx) is_sconnected(GD1);
       is_sconnected(GD2);
       is_sconnected(GD3);
```

Los grafos `GD1` y `GD2` son conexos. De ellos, sólo `GD1` es fuertemente conexo. El grafo `GD3` no es conexo, por lo que tampoco es fuertemente conexo.

Vamos a crear una función `ndg(G)` que se aplicará a un grafo `G` y que devolverá:

- `G`, si éste es no dirigido.
- El grafo (no dirigido) asociado a `G`, si éste es dirigido.

```
(%ixx) ndg(G) := create_graph(vertices(G),
                  listify(map(listify,map(setify,setify(edges(G))))));
```

Aplicamos la función recién creada a los ejemplos previos.

```
(%ixx) draw_graph(ndg(GD1), show_id=true);
       draw_graph(ndg(GD2), show_id=true);
       draw_graph(ndg(GD3), show_id=true);
```

Si la aplicamos a un grafo no dirigido, se obtiene ese mismo grafo.

```
(%ixx) draw_graph(K6, show_id=true);
       draw_graph(ndg(K6), show_id=true);

(%ixx) draw_graph(T1, show_id=true);
```

```
draw_graph(ndg(T1), show_id=true);
```

Por tanto podemos definir una función que se aplique a un grafo, ya sea dirigido o no, y que nos permita saber si tal grafo es o no conexo.

```
(%ixx) is_connected2(G):=is_connected(ndg(G));
```

Lo probamos con los grafos anteriores.

```
(%ixx) is_connected2(GD1);
       is_connected2(GD2);
       is_connected2(GD3);
       is_connected2(T1);
```

Recordemos que un grafo es plano si puede ser dibujado (en el plano) de modo que no se crucen sus lados. El comando `is_planar(G)` devuelve `true` si `G` es plano, y `false` en caso contrario. Sabemos que los grafos completos planos son exactamente K_1, K_2, K_3 y K_4.

Vamos a preguntarle cuáles de los ocho primeros grafos completos son planos.

```
(%ixx) makelist(is_planar(complete_graph(n)),n,1,8);
```

Como se puede apreciar, el resultado devuelto por Maxima coincide con lo que conocemos de teoría. Ahora vamos a ver cómo se obtiene una representación plana de un grafo plano.

```
(%ixx) K4:complete_graph(4);
       draw_graph(K4,redraw=true,program=planar_embedding);
```

Como K_4 es plano, para él sí se puede hacer dicha representación. Por contra, el grafo K_5 sabemos que no es plano.

```
(%ixx) K5:complete_graph(5);
       is_planar(K5);
```

Si ejecutamos el comando siguiente, da error:

```
(%ixx) draw_graph(K5,redraw=true,program=planar_embedding);
```

pues le estamos pidiendo que haga una representación plana de un grafo que no es plano.

Pero si le quitamos un lado a K_5, el grafo resultante sí es plano. Es un ejemplo de grafo plano maximal. Para ello, veamos antes cuales son los lados de K_5.

```
(%ixx) edges(K5);
```

Para suprimir un lado, sin suprimir los vértices incidentes con él, usamos el comando `remove_edge` que ya hemos comentado antes.

```
(%ixx) remove_edge([0,1],K5)$
```

```
(%ixx) draw_graph(K5, show_id=true, redraw=true,
                  program=planar_embedding);
```

Y vemos cómo nos hace una representación plana de este nuevo grafo.

A continuación vamos a construir un grafo que represente los movimientos del caballo en un tablero de ajedrez 4×4. Para ésto, numeramos las casillas desde 0 hasta 15. Leídas

de izquierda a derecha, las de la fila inferior van de 0 a 3. La fila siguiente de 4 a 7, y así sucesivamente. Dos casillas serán adyacentes, si el caballo puede saltar de una a otra. Tenemos entonces que el vértice 0 es adyacente con el 6 y con el 9. El vértice 1 es adyacente con los vértices 7, 8 y 10, etc.

El grafo que resulta lo almacenamos en la variable `ajedrez`:

```
(%ixx) ajedrez:create_graph(makelist(i,i,0,15),[[0,6],[0,9],[1,7],
[1,8],[1,10],[2,4],[2,9],[2,11],[3,5],[3,10],[4,10],[4,13],[5,11],[5,12],
[5,14],[6,8],[6,13],[6,15],[7,9],[7,14],[8,14],[9,15],[10,12],[11,13]]);
```

Sabemos que las casillas del tablero de ajedrez son o blancas o negras, y que un caballo, en cada movimiento, va de una blanca a una negra o viceversa. Esto nos dice que el grafo definido de esta forma es bipartido. Lo constatamos aplicándole al grafo el comando `is_bipartite`.

```
(%ixx) is_bipartite(ajedrez);
```

Es más, le podemos pedir que nos de la partición del conjunto de vértices en dos partes disjuntas.

```
(%ixx) bipartition(ajedrez);
```

Incluso, le vamos a decir que nos dibuje el grafo, separando los dos conjuntos de vértices.

```
(%ixx) [x,y]:bipartition(ajedrez)$

(%ixx) draw_graph(ajedrez, show_vertices=x, show_id=true);
```

Ejercicio. Compruebe mediante algunos comandos de Maxima que el grafo ciclo C_{15} no es bipartido, pero el grafo C_{16} sí lo es. □

El número cromático de un grafo se obtiene mediante el comando `chromatic_number`.

```
(%ixx) chromatic_number(K6);
       chromatic_number(K34);
       chromatic_number(CK34);
```

Recordemos de teoría que el número cromático de un grafo bipartido vale a lo sumo 2. Para nuestro grafo `ajedrez`,

```
(%ixx) chromatic_number(ajedrez);
```

Y si queremos una coloración del grafo, no tenemos más que escribir:

```
(%ixx) vertex_coloring(ajedrez);
```

El 2 que aparece en la primera posición de la respuesta de Maxima nos dice que es posible colorear el grafo con dos colores. Luego, a cada vértice le asigna un color (1 ó 2). La lista que aparece a continuación indica cada vértice y el color asignado.

Ejercicio. Escriba comandos en Maxima para obtener el número cromático y una coloración para el grafo ciclo C_{15}. □

Continuando con nuestro grafo `ajedrez`, podemos preguntarle por el camino más corto para ir del vértice 0 al 12.

```
(%ixx) shortest_path(0,12,ajedrez);
```

Ejercicio. Compruebe, mediante comandos apropiados de Maxima, si el grafo `ajedrez` tiene o no algún camino de Euler. □

Si escribimos el comando siguiente,

```
(%ixx) hamilton_cycle(ajedrez);
```

resulta la lista vacía, es decir, en el grafo `ajedrez` no hay ningún ciclo de Hamilton. Ello significa que un caballo de ajedrez no puede recorrer un tablero 4×4 sin repetir casillas y regresar a la casilla de origen.

```
(%ixx) hamilton_path(ajedrez);
```

Como resulta la lista vacía, vemos así que no es posible recorrer todo el tablero 4×4 pasando por cada casilla una sóla vez, es decir, en el grafo `ajedrez` no existen caminos abiertos de Hamilton.

Con la instrucción `girth(G)` obtenemos la longitud del ciclo más corto en un grafo `G`. Veamos el resultado para el grafo `ajedrez`.

```
(%ixx) girth(ajedrez);
```

Y con `odd_girth(G)` obtenemos la longitud del ciclo impar más corto en `G`. Introduzca el comando siguiente.

```
(%ixx) odd_girth(ajedrez);
```

El resultado obtenido significa que no hay ciclos de longitud impar. Esto, como sabemos de teoría, equivale a decir que el grafo es bipartido.

Otra forma de crear un grafo (no dirigido) en Maxima es con el comando `make_graph`. En este caso, hay que dar una lista o un conjunto de vértices, y una función booleana que se aplicará a dos vértices `u,v`, y valdrá `true` si ambos son adyacentes, mientras que valdrá `false` en caso contrario. Pruebe con el ejemplo siguiente.

```
(%ixx) f(x,y):=is(mod(x+y,3)=0)$
       H:make_graph(makelist(i,i,0,30),f);
```

En este grafo el conjunto de vértices está formado por los números naturales del 0 al 30. Además un vértice `u` será adyacente a otro `v` si `u+v` es múltiplo de 3.

Veamos que el grafo `H` que acabamos de definir, no es conexo.

```
(%ixx) is_connected(H);
```

De hecho, tiene dos componentes conexas. Una formada por los vértices que no son múltiplos de 3, y otra por los que sí son múltiplos de 3.

```
(%ixx) C:connected_components(H);
```

Ello es debido a que $x + y$ es múltiplo de 3, si y sólo si, x e y son ambos múltiplos de 3.

El comando `induced_subgraph` nos proporciona un subgrafo de un grafo. Para ello, hay que darle la lista de vértices que queremos considerar, y el grafo del que queremos extraer el subgrafo.

```
(%ixx) H1:induced_subgraph(C[1],H);

(%ixx) H2:induced_subgraph(C[2],H);
```

Con estos comandos hemos obtenido dos subgrafos de `H`, cada uno de ellos definido a partir de las listas de vértices almacenadas en `C`.

El subgrafo `H1` está formado por vértices de `H` que no son múltiplos de 3, con lo cual éstos son congruentes con 1 ó con 2 módulo 3. Nótese que los vértices congruentes con 1 módulo 3 están unidos a los que son congruentes con 2 módulo 3. Por tanto `H1` es un grafo bipartido.

```
(%ixx) is_bipartite(H1);
```

El subgrafo `H2` está formado por vértices de `H` que son múltiplos de 3.

Obtengamos la partición del conjunto de vértices de `H1`.

```
(%ixx) bipartition(H1);
```

De hecho, `H1` es bipartido completo pues es isomorfo a $K_{10,10}$.

```
(%ixx) is_isomorphic(H1,complete_bipartite_graph(10,10));
```

Y un isomorfismo podemos buscarlo con el comando `isomorphism` que ya vimos anteriormente.

```
(%ixx) K1010:complete_bipartite_graph(10,10)$

(%ixx) isomorphism(K1010,H1);
```

El grafo `H2` es un grafo completo. Todos los vértices que son múltiplos de 3 están unidos entre sí.

```
(%ixx) is_isomorphic(H2,complete_graph(11));
```

El comando `make_graph` también lo podemos usar para generar un grafo dirigido. Para ello hemos de incluir la opción `directed=true`. Pruebe los siguientes ejemplos.

```
(%ixx) g(x,y):=is(mod(x+y,3)=0)$
       M:make_graph(makelist(i,i,0,10), g, directed=true);
```

Le preguntamos por los lados y por los vértices del grafo generado.

```
(%ixx) vertices(M);
       edges(M);
```

Observe la diferencia si prescindimos de la opción `directed=true`.

```
(%ixx) g(x,y):=is(mod(x+y,3)=0)$
       M:make_graph(makelist(i,i,0,10), g);

(%ixx) vertices(M);
       edges(M);
```

Ejercicio. Defina cada uno de los grafos siguientes, para los que se especifica su conjunto de vértices y una condición que determinada la adyacencia entre sus vértices.

1. $G = (V, L)$ no dirigido, $V = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15\}$ y $u \sim v$ cuando $u + v$ es primo.

2. $G = (V, L)$ dirigido, $V = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15\}$ y $u \sim v$ cuando u divide a v.

3. $G = (V, L)$ dirigido, $V = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15\}$ y $u \sim v$ cuando u no divide a v.

4. $G = (V, L)$ no dirigido, $V = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \times \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ y $(a_1, b_1) \sim (a_2, b_2)$ cuando $a_1 + a_2 = b_1 + b_2$.

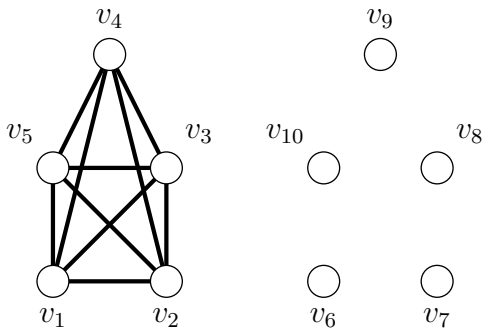
5. $G = (V, L)$ dirigido, $V = \mathcal{P}\big(\{1, 2, 3, 4\}\big)$ y $u \sim v$ cuando $u \subseteq v$.

6. $G = (V, L)$ no dirigido, $V = \mathcal{P}\big(\{1, 2, 3, 4\}\big)$ y $u \sim v$ cuando $|u| = |v|$.

7. $G = (V, L)$ no dirigido, $V = \mathcal{P}\big(\{1, 2, 3, 4\}\big)$ y $u \sim v$ cuando $u \cap \{0, 1, 4\} = v \cap \{0, 1, 4\}$.

8. $G = (V, L)$ no dirigido, $V = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15\}$ y $u \sim v$ cuando $u + v$ es múltiplo de 3 pero no es múltiplo de 6.

Ejercicio. Sea G el complementario del grafo siguiente:



Defina el grafo G en Maxima y a continuación mediante comandos apropiados responda a las cuestiones siguientes.

1. ¿Es G un grafo conexo?

2. Obtenga la secuencia de grados de G.

3. ¿Hay algún recorrido de Euler en G?

4. ¿Hay algún ciclo de Hamilton en G? ¿Y algún camino de Hamilton? Si la respuesta es afirmativa, muestre uno.

5. Calcule el número cromático de G.

6. ¿Es G un grafo bipartido?

7. Calcule el número de caminos de longitud 16 desde el vértice 0 hasta el vértice 9.

8. ¿Es G un grafo plano? Si la respuesta es negativa, ¿es posible suprimir un sólo lado de modo que el grafo resultante sí sea plano?

9. Obtenga el polinomio cromático de G.

□

Ejercicio. Defina un grafo `G` cuyos nodos vienen dados por los subconjuntos de dos elementos del conjunto $\{1, 2, 3, 4, 5, 6\}$, y dos nodos son adyacentes si y sólo si son disjuntos.

1. ¿Es `G` un grafo conexo?
2. Obtenga la secuencia de grados de `G`.
3. ¿Hay algún recorrido de Euler en `G`?
4. ¿Hay algún ciclo de Hamilton en `G`? ¿Y algún camino de Hamilton? Si la respuesta es afirmativa, muestre uno.
5. Calcule el número cromático de `G`.
6. ¿Es `G` un grafo bipartido?
7. Calcule el número de caminos de longitud 6 desde el vértice $\{1, 2\}$ hasta el vértice $\{5, 6\}$.
8. ¿Es `G` un grafo plano? Si la respuesta es negativa, ¿es posible suprimir un sólo lado de modo que el grafo resultante sí sea plano?

□