# Grafos

Maxima dispone de un paquete con operadores específicos para trabajar con grafos.

```
load(graphs)$
```

Puede trabajar con grafos simples (dirigidos o no dirigidos). Internamente los grafos son representados por sus listas de adyacencia.
Los vértices son identificados con números naturales y las aristas son representadas por listas de longitud dos. Se pueden asignar etiquetas a vértices y se pueden asignar pesos a las aristas para definir grafos ponderados.

El operador create_graph() sirve para definir una grafo a partir de su lista de vértices y su lista de aristas. Este operador admite varias sintaxis. En la más simple, escribimos un primer argumento con un número positivo n, que será el número de vértices y que serán identificados con los números 0, 1, 2,..., n-1. El segundo argumento es una lista de listas de longitud 2 que definen las aristas del grafo.

```
g1:create_graph(5,[[1,2],[1,3],[2,3],[0,4]]);
```

*GRAPH(5 vertices, 4 edges)*

Podemos ver la representación del grafo mediante su lista de adyacencia (escrita como columna):

```
print_graph(g1);
```

***Graph on 5 vertices with 4 edges.***
***Adjacencies:***
***4 : 0***
***3 : 2 1***
***2 : 3 1***
***1 : 3 2***
***0 : 4***

*done*

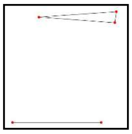
Y también podemos obtener la representación mediante la matriz de adyacencia del grafo:

```
mg1: adjacency_matrix(g1);
```

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

El operador draw_graph() implementa un algoritmo para construir una representación gráfica de los grafos.

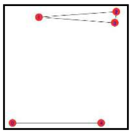
```
draw_graph(g1);
```



done

Este operador dispone de varias opciones para configurar la representación gráfica. Por ejemplo, con "show_id=true" conseguimos que aparezca la etiqueta de cada vértice y con "vertex_size=**" podemos aumentar el tamaño del círculo que los encierra.

```
draw_graph(g1,vertex_size=4,show_id=true);
```



done

También podemos elegir los números utilizados como vértices. Para ello, en el primer argumento del operador create_graph() escribimos la lista de los números que elijamos:

```
g2: create_graph([1,2,3,4,5,6,7,8], [[1,2], [1,6], [1,8],[1,3],
[2,4],[2,5],[2,7],[3,4],[3,5],[3,7],[4,6],[4,8],[5,6],[5,8],[6,7],[7,8]]);
```

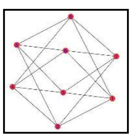
GRAPH(8 vertices, 16 edges)

```
print_graph(g2);
```

***Graph on 8 vertices with 16 edges.***
***Adjacencies:***
***1 : 3 8 6 2***
***2 : 7 5 4 1***
***3 : 7 5 4 1***
***4 : 8 6 3 2***
***5 : 8 6 3 2***
***6 : 7 5 4 1***
***7 : 8 6 3 2***
***8 : 7 5 4 1***

done

```
draw_graph(g2,vertex_size=3,show_id=true);
```



done

También podemos crear un grafo a partir de su matriz de adyacencia.
En ese caso, los vértices se etiquetarán con los números consecutivos desde el 0, siguiendo el orden dado por las filas y columnas.

```
mat:matrix([0,0,1,1,1],[0,0,1,0,1],[1,1,0,0,0],
    [1,0,0,0,0],[1,1,0,0,0]);
```

$$\begin{pmatrix} 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \end{pmatrix}$$

```
g3: from_adjacency_matrix(mat);
```

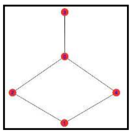
*GRAPH(5 vertices, 5 edges)*

```
print_graph(g3);
```

***Graph on 5 vertices with 5 edges.***
***Adjacencies:***
***4 : 1 0***
***3 : 0***
***2 : 1 0***
***1 : 4 2***
***0 : 4 3 2***

*done*

```
draw_graph(g3,vertex_size=4,show_id=true);
```



*done*

# 1 Propiedades básicas

Vamos a ver en las secciones siguientes los operadores disponibles para realizar las operaciones y analizar las propiedades que estamos estudiando en el curso.

La secuencia de grados de cada vértice la generamos con el operador degree_sequence()

```
degree_sequence(g1);
```

[1,1,2,2,2]

```
degree_sequence(g2);
```

[4,4,4,4,4,4,4,4]

```
degree_sequence(g3);
```

[1,2,2,2,3]

El operador is_conneceted() analiza si un grafo es o no conexo y connected_components() nos devuelve las componentes conexas.

```
is_connected(g1);
```

***false***

```
connected_components(g1);
```

[[2,1,3],[0,4]]

```
is_connected(g2);
```

***true***

```
connected_components(g2);
```

[[3,8,6,7,5,4,2,1]]

```
is_connected(g3);
```

***true***

```
connected_components(g3);
```

[[1,3,2,0,4]]

El operador is_bipartite() analiza si un grafo es o no bipartito y bipartition() determina las dos partes de un grafo si es bipartito y nos devuelve una lista vacía si no lo es.

```
is_bipartite(g1);
```

***false***

```
bipartition(g1);
```

[]

```
is_bipartite(g2);
```

***true***

```
bipartition(g2);
```

[[8,6,2,3],[7,5,4,1]]

```
is_bipartite(g3);
```

***true***

```
bipartition(g3);
```

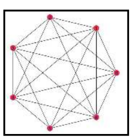
[[0,1],[4,3,2]]

# 2 *Ejemplos de grafos*

También disponemos de varios operadores para construir algunos tipos fundamentales de grafos (completos, ciclos, bipartitos,...)

## 2.1 Grafos completos

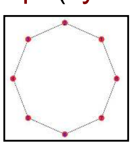
```
draw_graph(complete_graph(7),vertex_size=3,show_id=true);
```



*done*

## 2.2 Ciclos

```
draw_graph(cycle_graph(8),vertex_size=3,show_id=true);
```

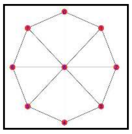


*done*

## 2.3 Grafos rueda

Obsérvese que grafo rueda 'n' tiene 'n+1' vértices:

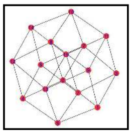
```
draw_graph(wheel_graph(8),vertex_size=3,show_id=true);
```



*done*

## 2.4 n-Cubos

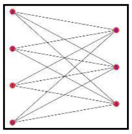
```
draw_graph(cube_graph(4),vertex_size=3,show_id=true);
```



*done*

## 2.5 Grafo bipartito completo

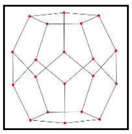
```
draw_graph(complete_bipartite_graph (4,3),vertex_size=3,show_id=true);
```



*done*

## 2.6 Dodecaedro

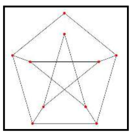
```
draw_graph(dodecahedron_graph());
```



*done*

## 2.7 Grafo de Petersen

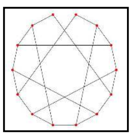
```
draw_graph(petersen_graph());
```



*done*

## 2.8 Grafo de Heawood

```
draw_graph(heawood_graph());
```



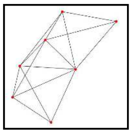
*done*

## 2.9 Grafos generados aleatoriamente

También disponemos de operadores para generar grafos de forma aleatoria.

random_graph(n,p) genera un grafo de n vértices, siendo p la probabilidad de que aparezca cada arista.

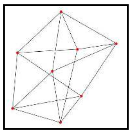
```
draw_graph(random_graph(7,.6));
```



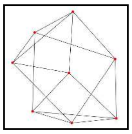
*done*

random_regular_graph(n,g) genera un grafo de n vértices cuyos grados son g

```
draw_graph(random_regular_graph(8,4));
draw_graph(random_regular_graph(8,4));
```



*done*



*done*

## 3 Isomorfismo de grafos

Disponemos del operador isomorphism() que analiza si dos grafos son isomorfos
y en tal caso determina un isomorfismo.

Los siguientes grafos tienen la misma secuencia gráfica:

```
g4: create_graph([1,2,3,4,5],[[1,4],[1,5],[2,3],
[2,4],[2,5],[3,4]])$
degree_sequence(g4);
```

[2,2,2,3,3]

```
g5: create_graph([1,2,3,4,5],[[1,2],[1,3],[1,5],
[2,3],[3,4],[4,5]])$
degree_sequence(g5);
```

[2,2,2,3,3]

```
g6: create_graph([1,2,3,4,5],[[1,4],[1,5],[2,4],
[2,5],[3,4],[3,5]])$
degree_sequence(g6);
```

[2,2,2,3,3]

g4 y g5 son isomorfos y el operador isomorphism() nos devuelve el isomorfismo:

```
isomorphism(g4,g5);
```

$[4 \to 3, 5 \to 5, 3 \to 2, 1 \to 4, 2 \to 1]$

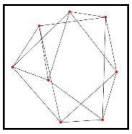
Sin embargo, g4 y g6 no son isomorfos y por eso, la salida de isomorphism() es una lista vacía:

```
isomorphism(g4,g6);
```

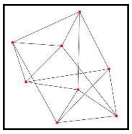
[ ]

En el siguiente ejemplo, generamos dos grafos 4-regulares con 8 vértices y podremos ver que en no todos son isomorfos.

```
g7: random_regular_graph(8,4)$
g8: random_regular_graph(8,4)$
draw_graph(g7);
draw_graph(g8);
isomorphism(g7,g8);
```



*done*



*done*

[ ]

# 4 Grafos hamiltonianos

El operador hamilton_cycle() determina si un grafo es o no Hamiltoniano y en tal caso calcula el ciclo de Hamilton.

```
k33: complete_bipartite_graph(3,3);
```

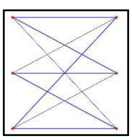
*GRAPH(6 vertices, 9 edges)*

```
hk33: hamilton_cycle(k33);
```

[0,5,2,4,1,3,0]

Utilizando la opción show_edges del operador draw_graph() podemos ver el dibujo de un grafo en el que se resalte un determinado camino, por ejemplo el ciclo de Hamilton de un grafo hamiltoniano.

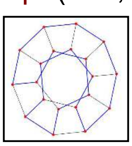
```
draw_graph(k33,show_edges=vertices_to_cycle(hk33));
```



*done*

En el siguiente ejemplo, vemos el ciclo de Hamilton contenido en el grafo determinado por los vértices y aristas de un dodecaedro.

```
dod: dodecahedron_graph()$
hdod: hamilton_cycle(dod)$
draw_graph(dod,show_edges=vertices_to_cycle(hdod));
```



*done*

También podemos determinar el camino de Hamilton contenido en un grafo semihamiltoniano. Por ejemplo, sabemos que el grafo K_3,4 no es hamiltoniano:

```
k34:complete_bipartite_graph(3,4)$
hamilton_cycle(k34);
```

[ ]

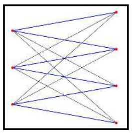
Pero K_3,4 sí es semihamiltonianto

```
hk34: hamilton_path(k34);
```

[6,2,5,1,4,0,3]

En este caso, el operador vertices_to_path() en la opción show_edges también nos permite visualizar el camino de Hamilton.

```
draw_graph(k34,show_edges=vertices_to_path(hk34));
```



*done*

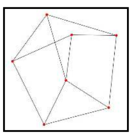
# 5 *Grafos planos*

Podemos analizar la planaridad de un grafo utilizanado el operador is_planar().

```
g9: create_graph(7, [[0,1],[0,5],[1,2],[1,4],
[2,3], [0,3], [2,5],[3,6],[4,5],[4,6],[5,6]])$
is_planar(g9);
```

***false***

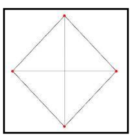
```
draw_graph(g9);
```



*done*

El algoritmo básico que utiliza Maxima para generar la representación gráfica de un grafo, no produce siempre una representación plana, como podemos ver con la representación de K4:

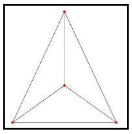
```
draw_graph(complete_graph(4));
```



*done*

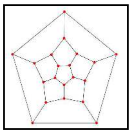
Sin embargo, Maxima permite elegir entre varios programas para determinar la posición final de los vértices en la representación gráfica. Por ejemplo, el programa "planar_embedding" fuerza la representación plana en la mayoría de las situaciones.

```
draw_graph(complete_graph(4),redraw=true,program=planar_embedding);
```



done

```
draw_graph(dodecahedron_graph(),redraw=true,program=planar_embedding);
```



done

# 6 Coloración

También disponemos de operadores que nos determinan el número cromático de un grafo y calculan una coloración óptima.

```
gcol:create_graph([1,2,3,4,5,6,7,8],[[3, 4],[4, 8],[2,5],
      [1,8],[5,6],[7,8],[4,7],[2,6],[1,4],[3,7],[2,7],[6,8],
      [2,3],[3,5],[1,6],[1,5]])$
```

```
chromatic_number(gcol);
```

4

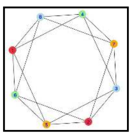
vertex_coloring(), nos devuelve también el número cromático pero también nos da una coloración con ese número de colores. Los colores también están representados por números naturales

```
vertex_coloring(gcol);
```

$$[4,[[8,2],[7,4],[6,1],[5,4],[4,1],[3,2],[2,3],[1,3]]]$$

Es decir, los vértices [4,6] están coloreados con el color 1, los vertices [8,3] con el color 2, los vértices [1,2] con el color 3 y los vértices [5,7] están coloreados con el color 4. Podemos visualizar el grafo con diferentes colores usando la opción vertex_partition.

```
draw_graph(gcol,vertex_size=4,show_id=true,
   vertex_partition=[[1,2],[3,8],[4,6],[5,7]]);
```



done

Sabemos que un grafo es bipartito si y solo si su número cromático es 2. Hemos visto que el operador bipartition() determina si un grafo es bipartito, devolviendo en tal caso los dos conjuntos de vértices. Por ejemplo, el grafo de Heawood que hemos visto anteriormente es bipartito y vamos a representarlo con mostrando una coloración con dos colores.

```
grhea: heawood_graph();
```

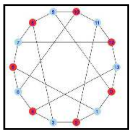
*GRAPH(14 vertices, 21 edges)*

```
parhea: bipartition(grhea);
```

$$[[8,12,6,10,0,2,4],[13,5,11,7,9,1,3]]$$

Con las opciones del operador draw_graph() podemos resaltar algunos vértices, lo que podemos usar para mostrar las partes de un grafo bipartito.

```
draw_graph(grhea,vertex_size=4,show_id=true,
vertex_partition=parhea);
```



*done*

# 7 Árboles

Un árbol es un grafo conexo que no contiene ciclos, el operador is_tree() analiza si un grafo es o no árbol.

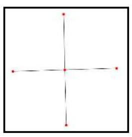
```
is_tree(cube_graph(3));
```

***false***

También podemos generar aleatoriamente árboles con un determinado número de vértices:

```
arb: random_tree(5)$
is_tree(arb);
draw_graph(arb);
```

***true***



*done*

Podemos determinar árboles generadores de un grafo, concretamente aquellos que contienen el menor número de aristas; para ello usamos el operador minimun_spanning_tree().

```
gt: create_graph([1,2,3,4,5,6,7],
[[1,2],
[1,6],
[2,3],
[2,4],
[2,6],
[2,7],
[3,4],
[4,5],
[4,7],
[5,6],
[5,7],
[6,7]]);
```

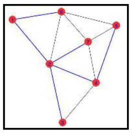
*GRAPH(7 vertices, 12 edges)*

```
sgt: minimum_spanning_tree(gt);
```

*GRAPH(7 vertices, 6 edges)*

El operador edges() usado dentro de la opción show_edges nos permite destacar las aristas de árbol generador de un grafo.

```
draw_graph(gt,vertex_size=4,show_id=true,show_edges=edges(sgt));
```



*done*

# 8 Grafos ponderados

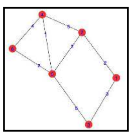
También disponemos de operadores para trabajar con grafos ponderados. Para definirlos, usamos también el operado create_graph, pero en este caso, las aristas se definirán con una lista de dos elementos, siendo el primero de ellos la arista propiamente dicha y el segundo el peso de la misma:

```
grp:create_graph([1,2,3,4,5,6],[
[[1,2],2],
[[1,3],3],
[[2,4],5],
[[2,5],2],
[[3,5],5],
[[4,5],1],
[[4,6],4],
[[5,6],2]
]);
```

*GRAPH(6 vertices, 8 edges)*

Para visualizar el peso de las aristas en la representación gráfica, usamos la opción show_weight:

```
draw_graph(grp,vertex_size=4,show_id=true,
    show_weight=true);
```



*done*

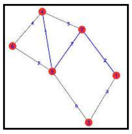
El algoritmo de Dijkstra determina el camino de longitud mínima entre dos vértices. Este algoritmo está implementado en el operador shortest_weighted_path(v1,v2,g) que determina el camino de peso mínimo en el grafo 'g' que une los vértices v1 y v2:

```
dijkgrp: shortest_weighted_path(1,4,grp);
```

$[5,[1,2,5,4]]$

Nuevamente, la opción show_edges nos permite visualizar el camino:

```
draw_graph(grp,vertex_size=4,show_id=true,
    show_weight=true,
    show_edges=vertices_to_path(dijkgrp[2]));
```



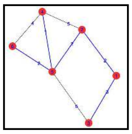
*done*

También podemos calcular árboles generadores minimales usando el operador minimun_spaning_tree()

```
mstgrp: minimum_spanning_tree(grp);
```

*GRAPH(6 vertices, 5 edges)*

```
draw_graph(grp,vertex_size=4,show_id=true,
   show_weight=true,
   show_edges=edges(mstgrp));
```



*done*