Dessin_Graphes

March 16, 2023

1 Dessiner des graphes

Marc Lorenzi

14 mars 2023

[1]:
```
import matplotlib.pyplot as plt
import math
import random
```

[2]:
```
plt.rcParams['figure.figsize'] = (6, 6)
```

1.1 1. Introduction

1.1.1 1.1 De quoi allons-nous parler ?

Nous allons dans ce notebook étudier un algorithme permettant de dessiner des graphes. Qu'est-ce qu'un *graphe* ? Qu'est-ce que *dessiner* ? Pourquoi **un** algorithme ? Commençons par le commencement.

1.1.2 1.2 Graphes non orientés

Soyons bien conscients qu'il existe un grand nombre de types de graphes : graphes, orientés, non orientés, étiquetés, etc. Nous allons nous concentrer dans la suite sur des graphes non orientés.

Définition. Un *graphe non orienté* est un couple $G = (S, A)$ où

- S est un ensemble fini, l'ensemble des *sommets* de G.
- A est une partie de $\mathcal{P}_2(S)$, l'ensemble des parties de S à deux éléments. Les éléments de A sont les *arêtes* de G.

Pour toute arête $A = \{x, y\}$ du graphe G, les sommets x et y sont les *extrémités* de l'arête a. Remarquons que $\{x, y\} = \{y, x\}$. C'est ici qu'intervient la non-orientation du graphe G.

Deux sommets x et y de G sont *voisins* (ou *adjacents*) s'ils sont les extrémités d'une arête de G.

Un graphe est un objet abstrait qui, malgré son nom, n'a rien de graphique. Il s'agit donc de définir ce que l'on entend par un *dessin* du graphe.

1.1.3 1.3 Dessins d'un graphe

Ici encore, il n'est pas facile de donner une définition universelle de ce que l'on entend par dessiner un graphe. Nous prendrons la définition suivante.

Définition. Soit $n \geq 2$. Soit $G = (S, A)$ un graphe. Un *dessin* de G est une fonction $D : S \longrightarrow \mathbb{R}^n$.

Bon, oui, d'accord, mais où est le dessin ? Patience, il arrive. Chaque sommet de G est associé par la fonction de dessin D à un point de $\mathbb{R}^n$. Nous avons donc *dessiné* les sommets de G. Il reste à dessiner les arêtes. Nous conviendrons dans la suite que si a est une arête de G d'extrémités $x, y \in S$, le dessin de l'arête A est le segment $[D(x), D(y)] \subset \mathbb{R}^n$. Nous pourrions faire des choses plus compliquées, comme par exemple dessiner les arêtes par des lignes courbes, des lignes brisées, etc. Restons raisonnables dans nos objectifs.

1.1.4 1.4 Pourquoi *un* algorithme ?

Une question plus pertinente serait : pourquoi voudrait-on dessiner un graphe ? Il y a de multiples réponses à cette question. L'une d'entre elles est notre désir de mettre en évidence certaines propriétés du graphe. Pour n'en citer que quelques-unes, on peut vouloir visualiser une *hiérarchie* dans les sommets, comme c'est le cas par exemple lorsque le graphe est un arbre. On peut également vouloir privilégier des regroupements de sommets, qui appartiennent à ce que l'on pourrait appeler des *clusters* (mot fourre-tout, extrêmement tendance). Etc, etc. Il est impossible de tout privilégier à la fois et il faut faire des choix. Ces choix sont dictés par le problème que l'on a à résoudre. Il est hors de question d'examiner tous les algorithmes existants, il y a des livres entiers sur le sujet. Nous allons de notre côté nous concentrer sur une technique de représentation des graphes, fondée sur les lois de la Physique. Pourquoi ? Pour deux raisons.

- Parce qu'elle donne souvent de bons résultats.
- Parce qu'elle est cool.

1.1.5 1.5 Représenter un graphe en Python

Encore et encore, il existe de multiples façons de modéliser les graphes dans un langage de programmation donné. Pour chaque problème de graphe, le choix de l'implémentation est crucial car c'est ce choix qui va déterminer la complexité de l'algorithme à coder. Nous allons choisir de représenter nos graphes par des *listes d'adjacence*.

Soit $G = (S, A)$ un graphe possédant n sommets. Décidons d'une énumération $x_0, \ldots, x_{n-1}$ des sommets de G. Ceci revient en gros à décider que $S = [|0, n-1|]$. Nous représentons le graphe G par une liste (que nous noterons encore G) de longueur n. Pour tout $i \in [|0, n-1|]$, $G[i]$ est aussi une liste. Précisément, c'est la liste des voisins de i dans le graphe G.

Pour des raisons d'efficacité, nous aurons aussi besoin de calculer la *matrice d'adjacence* d'un graphe G. Si l'ensemble des sommets de G est $S = \{x_0, \ldots, x_{n-1}\}$, c'est la matrice $A \in \mathcal{M}_n(\{0, 1\}$ définie pour tous $i, j \in [|0, n-1|]$ par $A_{ij} = 1$ si x_i et x_j sont voisins dans le graphe G et $A_{ij} = 0$ sinon.

La fonction `matrice` renvoie une matrice de taille $n \times n$ dont tous les coefficients sont égaux à x.

[3]:
```python
def matrice(n, x):
    A = n * [None]
    for i in range(n):
        A[i] = [x for j in range(n)]
    return A
```

La fonction `matrice_adjacence`prend en paramètre un graphe G. Elle renvoie la matrice d'adjacence de G.

[4]:
```python
def matrice_adjacence(G):
    n = len(G)
    A = matrice(n, 0)
    for i in range(n):
        for j in G[i]:
            A[i][j] = 1
    return A
```

Écrivons également une fonction `print_mat` qui affiche joliment une matrice d'adjacence.

[5]:
```python
def print_mat(A):
    n = len(A)
    s = n * '+---' + '+'
    for i in range(n):
        print(s)
        for j in range(n):
            print('| %1d ' % A[i][j], end='')
        print('|')
    print(s)
```

1.2 2. Quelques graphes

Donnons quelques exemples de graphes. Nous les dessinerons à la fin du notebook.

1.2.1 2.1 Graphes complets

Pour tout $n \in \mathbb{N}$, le *graphe complet* K_n a pour ensemble de sommets $S = [|0, n-1|]$. Pour tous $i, j \in [|0, n-1|]$ distincts, il y a une arête d'extrémités i et j.

[6]:
```python
def graphe_complet(n):
    G = n * [None]
    for i in range(n):
        G[i] = [j for j in range(n) if j != i]
    return G
```

[7]:
```python
graphe_complet(4)
```

[7]:
```
[[1, 2, 3], [0, 2, 3], [0, 1, 3], [0, 1, 2]]
```

[8]:
```
print_mat(matrice_adjacence(graphe_complet(4)))
```

```
+---+---+---+---+
| 0 | 1 | 1 | 1 |
+---+---+---+---+
| 1 | 0 | 1 | 1 |
+---+---+---+---+
| 1 | 1 | 0 | 1 |
+---+---+---+---+
| 1 | 1 | 1 | 0 |
+---+---+---+---+
```

1.2.2 2.2 Cycles

Pour tout $n \geq 3$, le *cycle* C_n a pour ensemble de sommets $S = [|0, n-1|]$. Pour tout $i \in [|0, n-1|]$, i a deux voisins qui sont $i-1$ et $i+1$ (modulo n).

[9]:
```
def graphe_cycle(n):
    G = n * [None]
    for i in range(n):
        G[i] = [(i - 1) % n, (i + 1) % n]
    return G
```

[10]:
```
graphe_cycle(4)
```

[10]:
```
[[3, 1], [0, 2], [1, 3], [2, 0]]
```

[11]:
```
print_mat(matrice_adjacence(graphe_cycle(4)))
```

```
+---+---+---+---+
| 0 | 1 | 0 | 1 |
+---+---+---+---+
| 1 | 0 | 1 | 0 |
+---+---+---+---+
| 0 | 1 | 0 | 1 |
+---+---+---+---+
| 1 | 0 | 1 | 0 |
+---+---+---+---+
```

1.2.3 2.3 Graphes bipartites complets

Pour tout $n \in \mathbb{N}$, le *graphe bipartite complet* $K_{n,n}$ a pour ensemble de sommets $S = [|0, 2n-1|]$. Pour tout $i \in [|0, n-1|]$ et tout $j \in [|n, 2n-1|]$, il y a une arête d'extrémités i et j.

[12]:
```
def graphe_bipartite_complet(n):
    G = (2 * n) * [None]
    for i in range(n):
        G[i] = [j for j in range(n, 2 * n)]
    for i in range(n, 2 * n):
        G[i] = [j for j in range(n)]
    return G
```

[13]:
```
print_mat(matrice_adjacence(graphe_bipartite_complet(3)))
```

```
+---+---+---+---+---+---+
| 0 | 0 | 0 | 1 | 1 | 1 |
+---+---+---+---+---+---+
| 0 | 0 | 0 | 1 | 1 | 1 |
+---+---+---+---+---+---+
| 0 | 0 | 0 | 1 | 1 | 1 |
+---+---+---+---+---+---+
| 1 | 1 | 1 | 0 | 0 | 0 |
+---+---+---+---+---+---+
| 1 | 1 | 1 | 0 | 0 | 0 |
+---+---+---+---+---+---+
| 1 | 1 | 1 | 0 | 0 | 0 |
+---+---+---+---+---+---+
```

1.2.4 2.4 Les graphes platoniciens

Il existe 5 polyèdres réguliers : le tétraèdre, le cube, l'octaèdre, le dodécaèdre et l'icosaèdre. On peut associer à chacun de ces sommets un graphe « abstrait » dont les sommets sont les sommets du polyèdre et les arêtes sont les … arêtes du polyèdre.

Inutile de nous attarder sur le tétraèdre, son graphe est le graphe complet K_4. Nous passons aussi sur le cube parce que nous parlerons un peu plus loin des *hypercubes*.

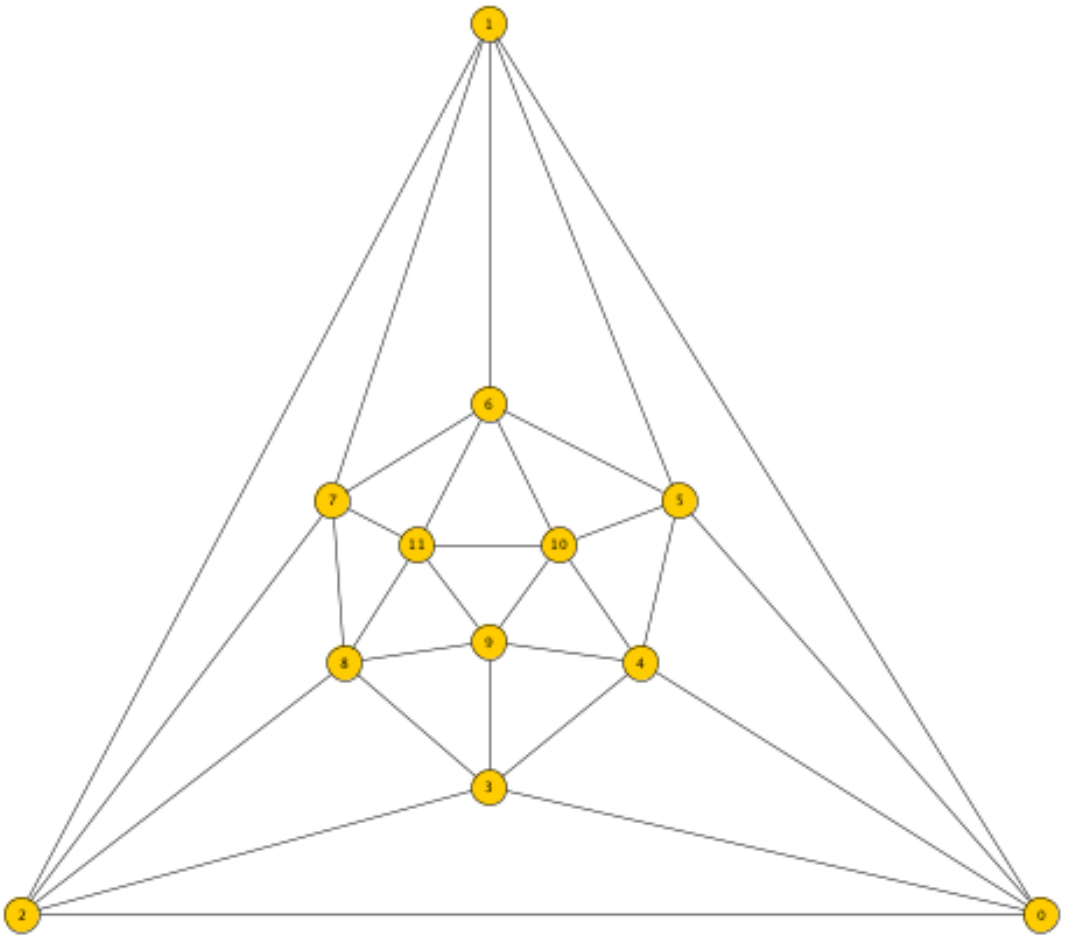
Ci-dessous, les graphes associés aux trois autres polyèdres réguliers.

[14]:
```
def graphe_octaedre():
    G = 6 * [None]
    G[0] = [1, 3, 4, 5]
    G[1] = [0, 2, 4, 5]
    G[2] = [1, 3, 4, 5]
    G[3] = [0, 2, 4, 5]
    G[4] = [0, 1, 2, 3]
    G[5] = [0, 1, 2, 3]
    return G
```

Voici la matrice d'adjacence de l'octaèdre.

[15]:
```
print_mat(matrice_adjacence(graphe_octaedre()))
```

```
+---+---+---+---+---+---+
| 0 | 1 | 0 | 1 | 1 | 1 |
+---+---+---+---+---+---+
| 1 | 0 | 1 | 0 | 1 | 1 |
+---+---+---+---+---+---+
| 0 | 1 | 0 | 1 | 1 | 1 |
+---+---+---+---+---+---+
| 1 | 0 | 1 | 0 | 1 | 1 |
+---+---+---+---+---+---+
| 1 | 1 | 1 | 1 | 0 | 0 |
+---+---+---+---+---+---+
| 1 | 1 | 1 | 1 | 0 | 0 |
+---+---+---+---+---+---+
```

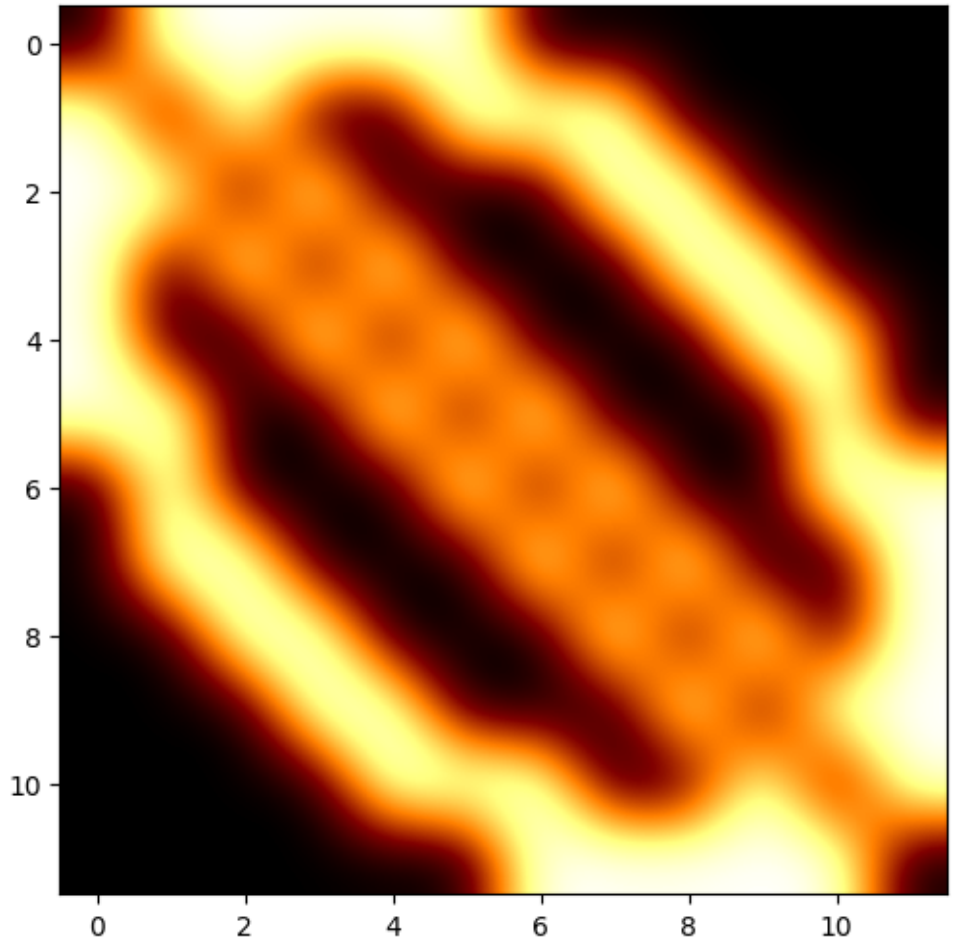


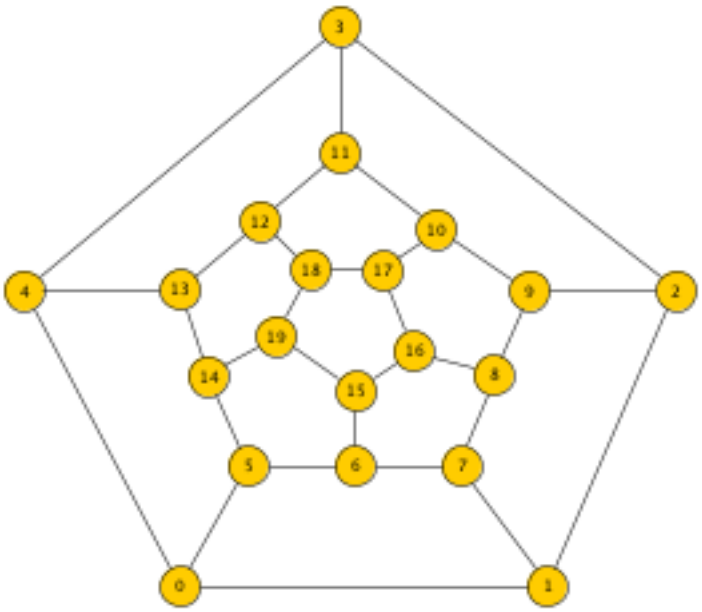
[16]:
```
def graphe_icosaedre():
    G = 12 * [None]
    G[0] = [1, 2, 3, 4, 5]
    G[1] = [0, 2, 5, 6, 7]
    G[2] = [0, 1, 3, 7, 8]
```

```
    G[3] = [0, 2, 4, 8, 9]
    G[4] = [0, 3, 5, 9, 10]
    G[5] = [0, 1, 4, 6, 10]
    G[6] = [1, 5, 7, 10, 11]
    G[7] = [1, 2, 6, 8, 11]
    G[8] = [2, 3, 7, 9, 11]
    G[9] = [3, 4, 8, 10, 11]
    G[10] = [4, 5, 6, 9, 11]
    G[11] = [6, 7, 8, 9, 10]
    return G
```

[17]:
```
A = matrice_adjacence(graphe_icosaedre())
plt.imshow(A, cmap='afmhot', interpolation='bicubic')
```

[17]: <matplotlib.image.AxesImage at 0x11a7121a0>

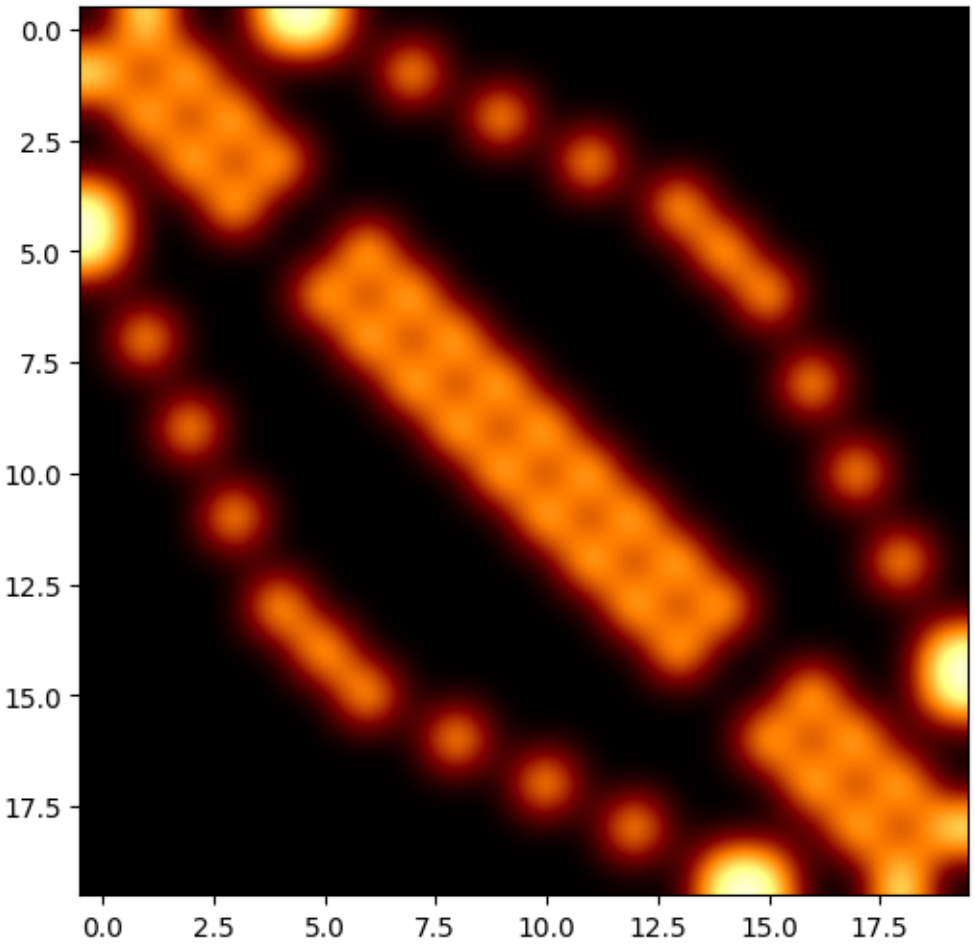




[18]:
```
def graphe_dodecaedre():
    G = 20 * [None]
    G[0] = [1, 4, 5]
    G[1] = [0, 2, 7]
    G[2] = [1, 3, 9]
    G[3] = [2, 4, 11]
    G[4] = [0, 3, 13]
    G[5] = [0, 6, 14]
    G[6] = [5, 7, 15]
    G[7] = [1, 6, 8]
    G[8] = [7, 9, 16]
    G[9] = [2, 8, 10]
    G[10] = [9, 11, 17]
    G[11] = [3, 10, 12]
    G[12] = [11, 13, 18]
    G[13] = [4, 12, 14]
    G[14] = [5, 13, 19]
    G[15] = [6, 16, 19]
    G[16] = [8, 15, 17]
    G[17] = [10, 16, 18]
    G[18] = [12, 17, 19]
    G[19] = [14, 15, 18]
    return G
```

[19]:
```
A = matrice_adjacence(graphe_dodecaedre())
plt.imshow(A, cmap='afmhot', interpolation='bicubic')
```

[19]:
```
<matplotlib.image.AxesImage at 0x11afefa90>
```



1.2.5 2.7 Le graphe de Petersen

Le *graphe de Petersen* est un graphe qui possède de nombreuses propriétés intéressantes.

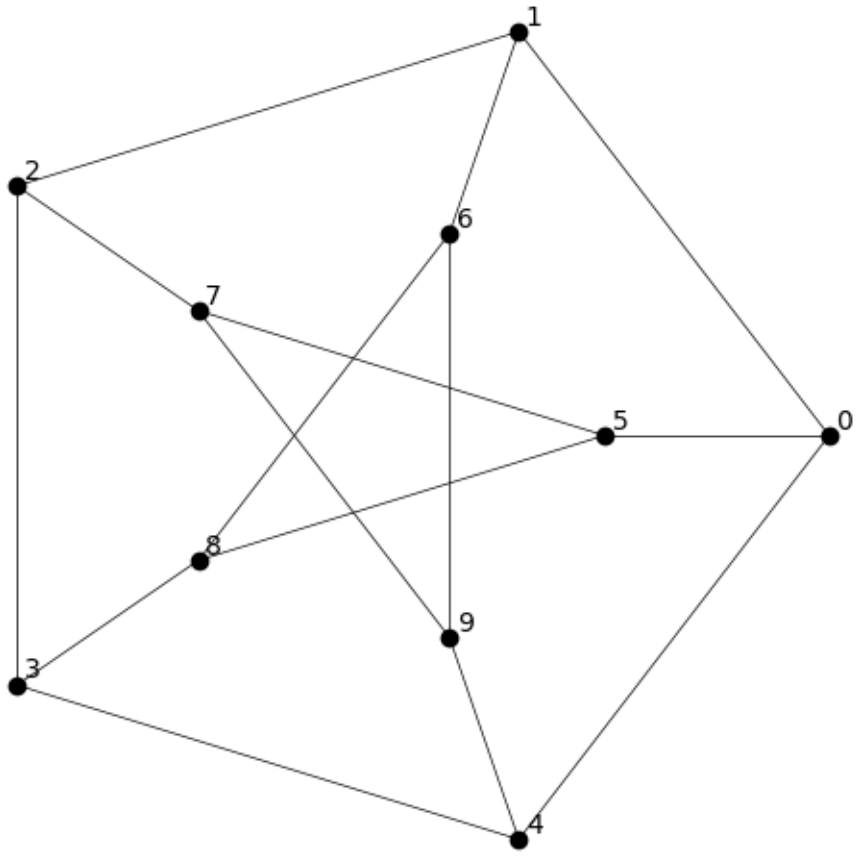
[20]:
```
def graphe_petersen():
    G = 10 * [None]
    G[0] = [1, 4, 5]
    G[1] = [0, 2, 6]
    G[2] = [1, 3, 7]
    G[3] = [2, 4, 8]
    G[4] = [0, 3, 9]
    G[5] = [0, 7, 8]
    G[6] = [1, 8, 9]
    G[7] = [2, 5, 9]
    G[8] = [3, 5, 6]
    G[9] = [4, 6, 7]
```

```
    return G
```

En voici une représentation graphique.

[21]:
```
def plot_petersen():
    xs = [2 * math.cos(2 * k * math.pi / 5) for k in range(5)]
    ys = [2 * math.sin(2 * k * math.pi / 5) for k in range(5)]
    plt.plot(xs, ys, 'ok')
    for k in range(5):
        plt.plot([xs[k], xs[(k + 1) % 5]], [ys[k], ys[(k + 1) % 5]], 'k', lw=0.5)
        plt.text(xs[k] + 0.03, ys[k] + 0.03, str(k))
    xs1 = [x / 2 for x in xs]
    ys1 = [y / 2 for y in ys]
    plt.plot(xs1, ys1, 'ok')
    for k in range(5):
        plt.plot([xs1[k], xs1[(k + 2) % 5]], [ys1[k], ys1[(k + 2) % 5]], 'k',␣
 ↪lw=0.5)
        plt.text(xs1[k] + 0.03, ys1[k] + 0.03, str(k + 5))
    for k in range(5):
        plt.plot([xs[k], xs1[k]], [ys[k], ys1[k]], 'k', lw=0.5)
    plt.axis('off')
```

[22]:
```
plot_petersen()
```



1.2.6 2.8 Hypercubes

Pour tout $n \in \mathbb{N}$, le *cube* Q_n a pour ensemble de sommets $S = [|0, 2^n - 1|]$. Deux sommets i et j sont voisins dans Q_n lorsque ils diffèrent dans leur développement en base 2 par un seul bit. Pour $n = 3$, on retrouve le cube classique.

La fonction `vers_bits` prend en paramètres deux entiers naturels x et n, où $x < 2^n$. L'entier x s'écrit de façon unique

$$x = \sum_{k=0}^{n-1} b_k 2^k$$

où les b_k valent 0 ou 1. La fonction renvoie la liste $[b_0, \dots, b_{n-1}]$.

[23]:
```
def vers_bits(x, n):
    s = []
    for k in range(n):
        s.append(x % 2)
```

```
        x = x // 2
    return s
```

[24]:
```
vers_bits(24, 5)
```

[24]:
```
[0, 0, 0, 1, 1]
```

La fonction `vers_entier` fait l'opération inverse. Elle prend en paramètre une liste $[b_0, \dots, b_{n-1}]$ de bits et renvoie l'entier $\sum_{k=0}^{n-1} b_k 2^k$.

[25]:
```
def vers_entier(bits):
    x = 0
    n = len(bits)
    for k in range(n):
        x += bits[k] * 2 ** k
    return x
```

[26]:
```
vers_entier([0, 0, 0, 1, 1])
```

[26]:
```
24
```

Voici la fonction renvoyant le graphe de l'hypercube Q_n.

[27]:
```
def graphe_hypercube(n):
    p2n = 2 ** n
    G = p2n * [None]
    for i in range(p2n):
        G[i] = []
        bs = vers_bits(i, n)
        for k in range(n):
            bs1 = bs.copy()
            bs1[k] = 1 - bs1[k]
            j = vers_entier(bs1)
            G[i].append(j)
    return G
```

Et voici la matrice d'adjacence de Q_3.

[28]:
```
print_mat(matrice_adjacence(graphe_hypercube(3)))
```

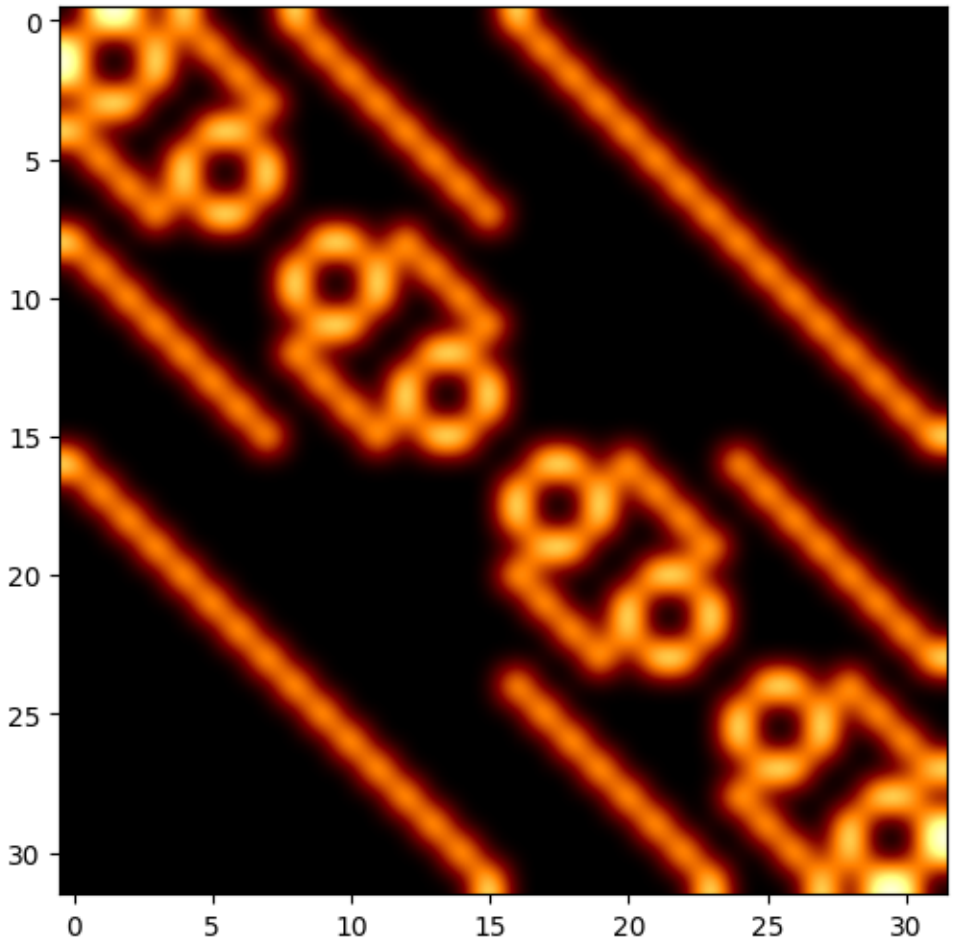
```
+---+---+---+---+---+---+---+---+
| 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 |
+---+---+---+---+---+---+---+---+
| 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
+---+---+---+---+---+---+---+---+
| 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
+---+---+---+---+---+---+---+---+
| 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 |
```

```
+---+---+---+---+---+---+---+---+
| 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
+---+---+---+---+---+---+---+---+
| 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 |
+---+---+---+---+---+---+---+---+
| 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 |
+---+---+---+---+---+---+---+---+
| 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 |
+---+---+---+---+---+---+---+---+
```

Voici également une représentation sympa de la matrice d'adjacence de Q_5.

[29]:
```
plt.imshow(matrice_adjacence(graphe_hypercube(5)), cmap='afmhot',␣
↪interpolation='bicubic')
```

[29]:
```
<matplotlib.image.AxesImage at 0x11b2bcf10>
```



1.3 3. Forces

1.3.1 3.1 L'idée

L'idée de l'algorithme de dessin de graphe que nous allons écrire est la suivante. Soit G un graphe. Choisissons une position initiale « aléatoire » des sommets de G. Chaque sommet de G est identifié à un point matériel. Ce point est soumis à des forces exercées par les autres sommets. Dans l'algorithme que nous avons choisi d'écrire, ces forces sont de deux types.

- Une force répulsive : tous les sommets du graphe ont tendance à se repousser les uns les autres.
- Une force attractive : deux sommets *voisins* du graphe ont tendance à s'attirer mutuellement.

Une position d'équilibre du système formé par l'ensemble des sommets sera, on l'espère, telle que des sommets reliés par des arêtes soient proches, alors que des sommets qui ne sont pas voisins dans le graphe soient éloignés.

Il n'y a plus qu'à coder, mais avant parlons un peu de vecteurs.

1.3.2 3.2 Vecteurs

La classe `Vecteur` permet de d'opérer de façon naturelle sur des vecteurs.

[30]:
```
class Vecteur:

    def __init__(self, pos): self.pos = pos

    def __len__(self): return len(self.pos)
    def __repr__(self): return str(self.pos)
    def __str__(self): return str(self.pos)
    def __getitem__(self, i): return self.pos[i]
    def __setitem__(self, i, x): self.pos[i] = x

    def __add__(self, y):
        n = len(self)
        return Vecteur([self[i] + y[i] for i in range(n)])

    def __neg__(self):
        n = len(self)
        return Vecteur([-self[i] for i in range(n)])

    def __sub__(self, y): return self + (-y)

    def __mul__(self, y):
        n = len(self)
        s = 0
        for i in range(n):
            s += self[i] * y[i]
```

```
        return s

    def __rmul__(self, t):
        n = len(self)
        return Vecteur([t * self[i] for i in range(n)])

    def norme(self): return math.sqrt(self * self)
```

Voici quelques exemples d'utilisation de cette classe. Tout d'abord, créons deux vecteurs x et y.

[31]:
```
x = Vecteur([1, 2])
y = Vecteur([3, 4])
```

Addition, soustraction.

[32]:
```
print(x + y)
print(x - y)
```

```
[4, 6]
[-2, -2]
```

Produit scalaire.

[33]:
```
print(x * y)
```

```
11
```

Produit par un scalaire.

[34]:
```
print(3 * x)
```

```
[3, 6]
```

Et enfin, norme.

[35]:
```
print(y.norme())
```

```
5.0
```

Nous aurons besoin de créer des vecteurs aléatoires. La fonction `vecteur_alea` crée un vecteur dont les coordonnées sont des réels aléatoires de l'intervalle $[-1, 1]$.

[36]:
```
def vecteur_alea(dim):
    return Vecteur([random.uniform(-1, 1) for k in range(dim)])
```

[37]:
```
print(vecteur_alea(4))
```

```
[-0.6548635468736344, 0.12737723760483322, 0.8890147969751179,
0.6957162446767491]
```

La fonction `liste_vecteurs_alea` renvoie une liste de vecteurs aléatoires.

[38]:
```
def liste_vecteurs_alea(dim, n):
    return [vecteur_alea(dim) for k in range(n)]
```

Voici une liste de 5 vecteurs aléatoires en dimension 2.

[39]:
```
liste_vecteurs_alea(2, 5)
```

[39]:
```
[[-0.5036488737000244, -0.6292613803118914],
 [-0.9520065832195512, 0.576931437551401],
 [0.4097535428223946, 0.28624677112062735],
 [0.28754118186509814, 0.17559588190367226],
 [-0.091812459199484, 0.8057744315438575]]
```

1.3.3 3.3 Attraction et répulsion

Soient $x, y \in \mathbb{R}^2$, censés représenter les positions de deux sommets d'un graphe G. Posons

$$u = \frac{\overrightarrow{xy}}{||\overrightarrow{xy}||}$$

Le point y exerce une force de *répulsion* sur le point x donnée par

$$f_{rep} = -\frac{\ell^2}{||\overrightarrow{xy}||}u$$

où ℓ est un paramètre qui contrôle l'intensité de la force.

Remarquons que le point x exerce sur le point y une force qui est l'opposé de la force ci-dessus. Newton serait content de nous. Action, réaction.

[40]:
```
def repulsion(x, y, L=1):
    z = y - x
    d = z.norme()
    u = (1 / d) * z
    return (-L ** 2 / d) * u
```

[41]:
```
x = Vecteur([1, 2])
y = Vecteur([3, 5])
print(repulsion(x, y))
```

```
[-0.15384615384615385, -0.23076923076923078]
```

Si x et y sont les positions de deux sommets *voisins* de G, le point y exerce également une force *d'attraction* sur le point x, donnée par

$$f_{attr} = \frac{||\overrightarrow{xy}||^2}{\ell}u$$

[42]:
```
def attraction(x, y, L=1):
    z = y - x
    d = z.norme()
    u = (1 / d) * z
    return (d ** 2 / L) * u
```

[43]:
```
x = Vecteur([1, 2])
y = Vecteur([3, 5])
print(attraction(x, y))
```

```
[7.211102550927978, 10.816653826391967]
```

Si nous posons $d = ||\overrightarrow{xy}||$, nous avons donc, lorsque x et y sont voisins dans le graphe G, une force totale exercée sur x par y égale à

$$f = f_{rep} + f_{attr} = \left(-\frac{\ell^2}{d} + \frac{d^2}{\ell}\right) u = \frac{d^3 - \ell^3}{\ell d} u$$

Ainsi, puisque u est unitaire, l'intensité algébrique de la force subie par x est

$$\phi(d) = \frac{d^3 - \ell^3}{\ell d}$$

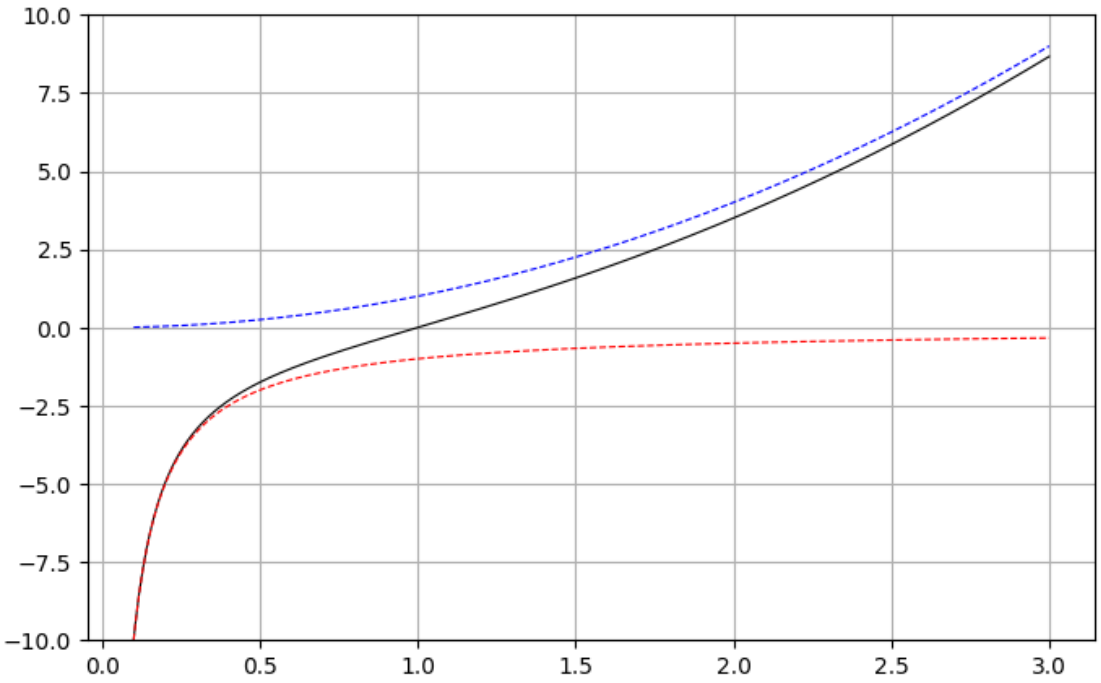
Par algébrique, nous entendons que la force est totale est attractive si $\phi(d) > 0$ et répulsive sinon. On a donc attraction si et seulement si $d > \ell$. Illustrons tout cela.

[44]:
```
def phi(d, L=1): return (d ** 3 - L ** 3) / (L * d)
```

[45]:
```
plt.rcParams['figure.figsize'] = (8, 5)
```

[46]:
```
def plot_phi():
    dmin = 0.1
    dmax = 3
    ds = [dmin + k * (dmax - dmin) / 500 for k in range(501)]
    fs = [phi(d) for d in ds]
    gs = [- 1 / d for d in ds]
    hs = [d ** 2 for d in ds]
    plt.plot(ds, fs, 'k', lw=0.8)
    plt.plot(ds, gs, '--r', lw=0.8)
    plt.plot(ds, hs, '--b', lw=0.8)
    plt.ylim(-10, 10)
    plt.grid()
```

[47]:
```
plot_phi()
```



En rouge, la force répulsive. En bleu, la force attractive. En noir, la force totale. Comprenons bien que deux sommets reliés par une arête exercent chacun sur l'autre la force noire. Si au contraire, les sommets ne sont pas voisins, chacun exerce sur l'autre la force répulsive rouge.

[48]:
```
plt.rcParams['figure.figsize'] = (6, 6)
```

1.3.4 3.4 Itérations

La fonction `step` prend en paramètres

- une matrice A, censée être la matrice d'adjacence d'un graphe G.
- une liste de vecteurs qui sont les positions courantes des sommets de G.
- un paramètre optionnel δ, que l'on peut considérer comme un petit intervalle de temps dt.

Pour chaque sommet du graphe G, la fonction calcule la somme f des forces exercées sur ce sommet par les autres sommets du graphe. Elle modifie ensuite la position du sommet de la quantité

$$dx = \delta f$$

Comme aucun physicien ne nous regarde, rien ne nous empêche d'additionner des positions et des vitesses. La fonction renvoie la nouvelle liste de positions.

[49]:
```
def step(A, positions, delta=0.01):
    n = len(positions)
    dim = len(positions[0])
```

```
    forces = n * [None]
    for i in range(n):
        forces[i] = Vecteur(dim * [0])
        for j in range(n):
            if j != i:
                forces[i] = forces[i] + repulsion(positions[i], positions[j])
                if A[i][j] == 1:
                   forces[i] = forces[i] + attraction(positions[i], positions[j])
    positions1 = n * [None]
    for i in range(n):
        positions1[i] = positions[i] + delta * forces[i]
    return positions1
```

Voici la fonction principale. On calcule la matrice d'adjacence du graphe G et on se donne une liste de positions aléatoires pour les sommets de G. On itère ensuite `niter` fois la fonction `step`, et on renvoie la liste finale de positions.

[50]:
```
def iterer(G, dim, niter, delta=0.01):
    n = len(G)
    A = matrice_adjacence(G)
    positions = liste_vecteurs_alea(dim, n)
    for k in range(niter):
        positions = step(A, positions)
    return positions
```

[51]:
```
iterer(graphe_complet(5), 2, 100)
```

[51]:
```
[[0.42366751069047176, -0.17358486307380905],
 [0.12943531881782216, 0.4880155246310542],
 [-0.7370914906608534, -0.30192540323273015],
 [-0.10960842046497728, -0.666075574939701],
 [-0.5948634962899233, 0.4075405288906576]]
```

1.3.5 3.5 Dessiner, enfin !

La fonction `plot_graphe` se passe de commentaire. Elle dessine le graphe G après avoir effectué `niter` itérations de notre algorithme.

[52]:
```
def plot_graphe(G, niter=1000, delta=0.1):
    positions = iterer(G, 2, niter, delta)
    xs = [A[0] for A in positions]
    ys = [A[1] for A in positions]
    n = len(G)
    for i in range(n):
        for j in G[i]:
            A = positions[i]
```

```
            B = positions[j]
            plt.plot([A[0], B[0]], [A[1], B[1]], 'k', lw=0.4)
    plt.plot(xs, ys, 'ok', ms=10)
    plt.axis('off')
    plt.savefig('graphe.png', bbox_inches='tight')
```

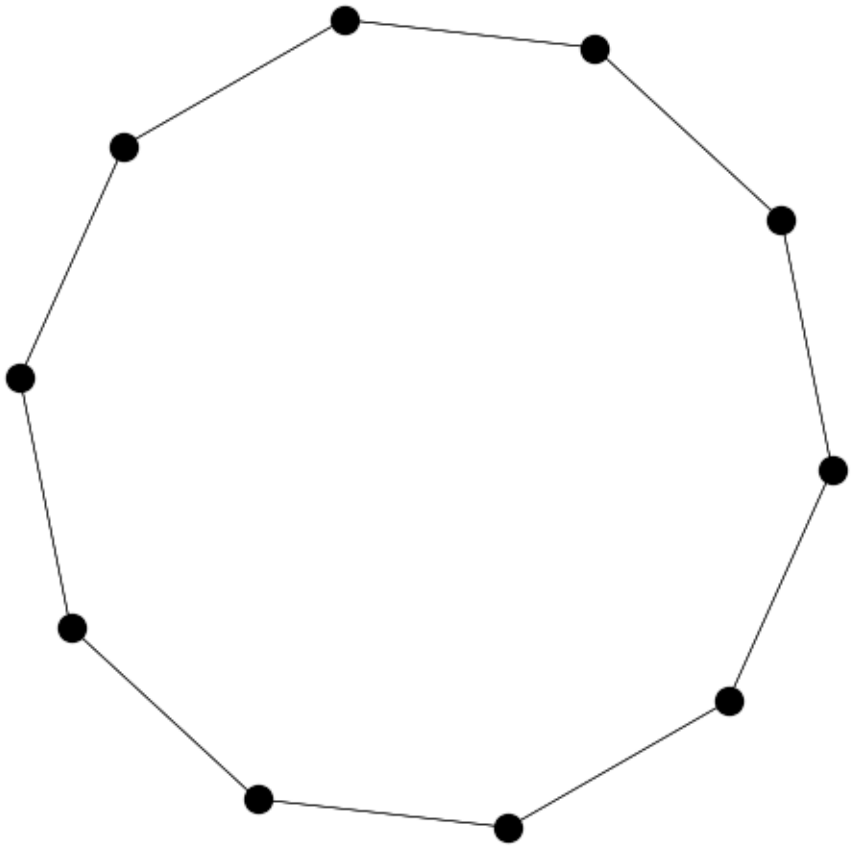
Il ne reste plus qu'à tester. Remarquons toutefois qu'il peut arriver que le résultat soit décevant. Augmenter la valeur du nombre d'itérations `niter` peut régler le problème. On peut aussi réévaluer la cellule, cela peut suffire.

Il est également possible que la valeur par défaut $\delta = 0.1$ ne soit pas bien adaptée.

Une petite remarque avant de nous lancer : chaque itération de notre algorithme a une complexité en $O(n^2)$, où n est le nombre de sommets du graphe. Nous éviterons donc d'exécuter `plot_graphe` avec des graphes ayant 1000 sommets !

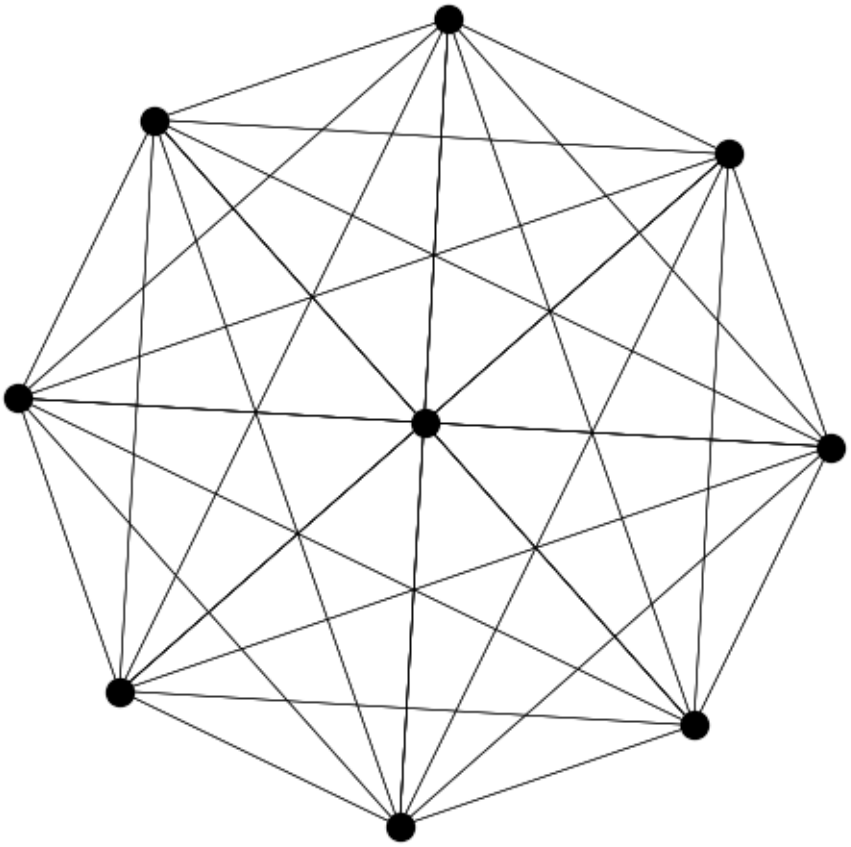
Commençons par un cycle de longueur 5.

[53]:
```
plot_graphe(graphe_cycle(10))
```



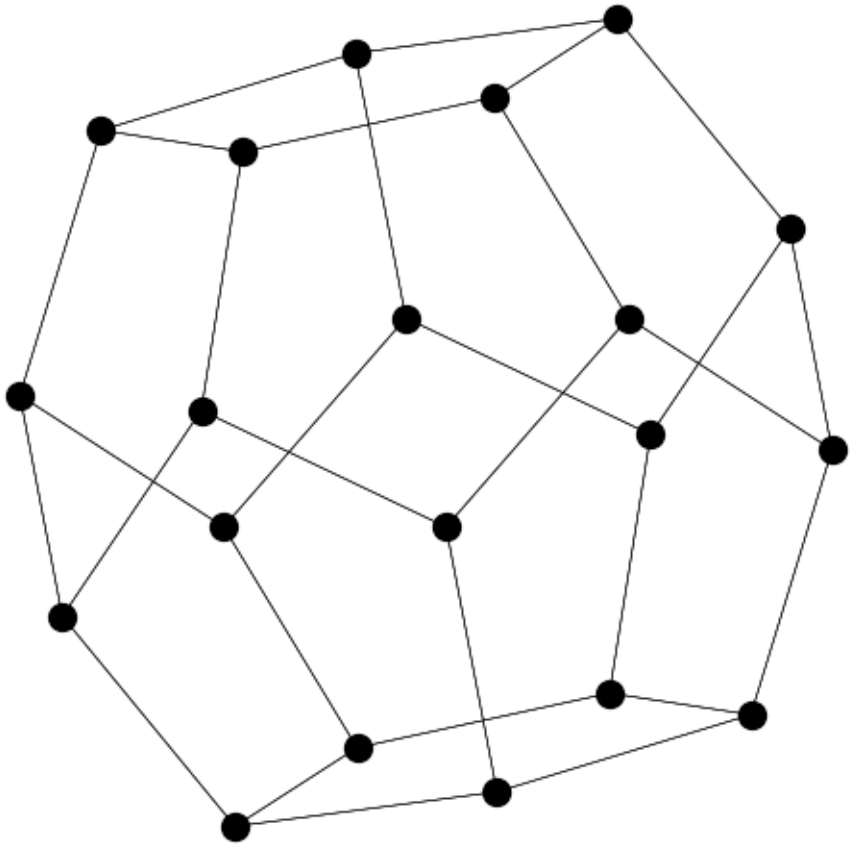
Puis un graphe complet.

[54]:
```
plot_graphe(graphe_complet(9))
```



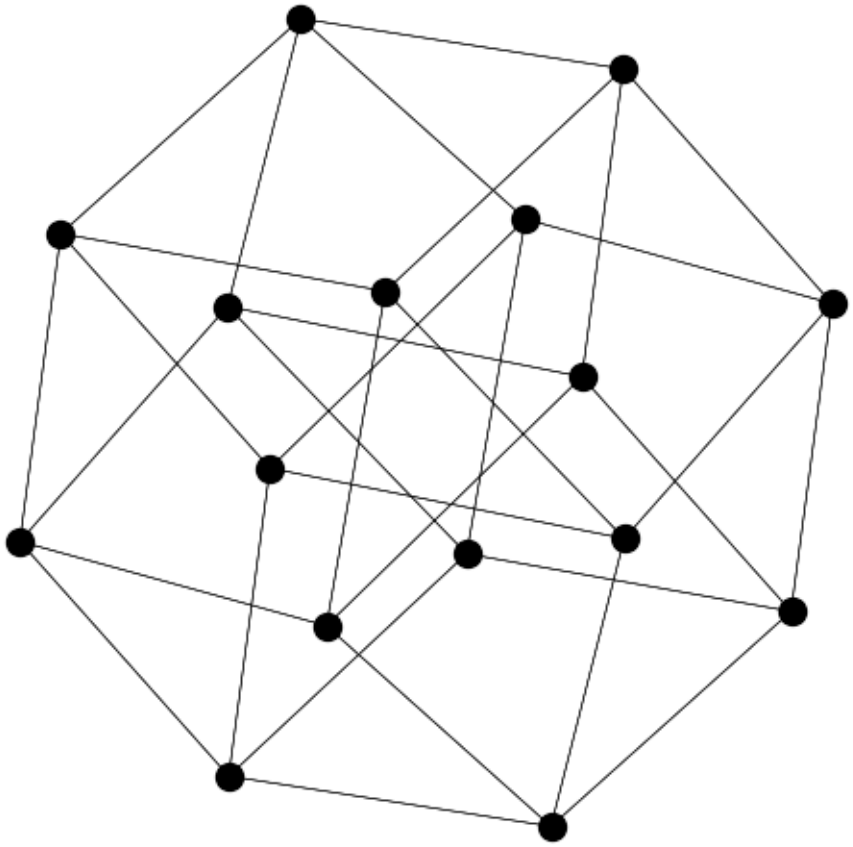
Et un dodécaèdre.

[57]:
```
plot_graphe(graphe_dodecaedre())
```



Finissons par l'hypercube Q_4.

[56]:
```
plot_graphe(graphe_hypercube(4))
```



Amusez-vous bien avec les autres graphes …

Bibliographie

- M. Kaufmann, D. Wagner (Eds.), *Drawing Graphs, Methods and Models*, Springer (2001)
- G. Di Battista, P. Eades, R. Tamassia, I.G. Tollis, *Graph Drawings, Algorithms for the Visualization of Graphs*, Prentice Hall (1999)
- R. Tamassia, *Handbook of Graph Drawing and Visualization*, CRC Press (2013)