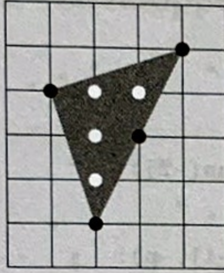
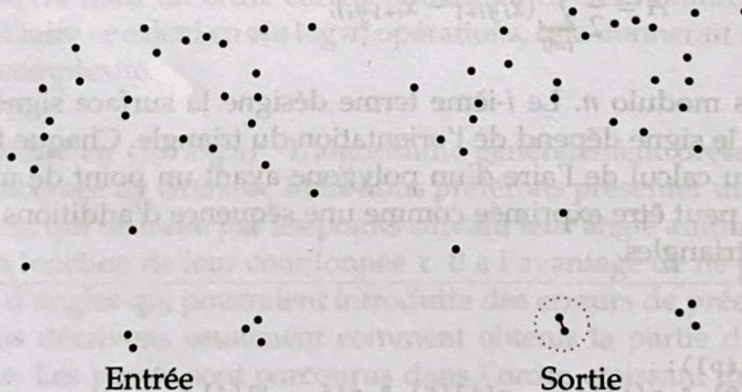


FIGURE 12.2 – Le nombre de points entiers sur le contour (en noir) est 4, le nombre de points entiers intérieurs (en blanc) est 4 et la surface est $4 + 4/2 - 1 = 5$.

Calcul du nombre de points entiers intérieurs Ce nombre est obtenu par le théorème de Pick, qui relie la surface A d'un polygone, le nombre de points intérieurs n_i et le nombre de points n_b sur le contour du polygone par l'équation

$$A = n_i + \frac{n_b}{2} - 1.$$

12.3 Paire de points les plus proches



Application Des tentes sont disposées arbitrairement dans un camping, chacune est occupée par quelqu'un qui possède un poste de radio. On souhaite imposer un seuil de volume commun à tous les campeurs à ne pas dépasser pour que personne ne soit gêné par la musique d'un de ses voisins.

Définition Étant donné n points $p_1, \dots, p_n$, on souhaite déterminer une paire de points p_i, p_j qui minimise la distance euclidienne entre p_i et p_j.

Algorithme randomisé en temps linéaire Plusieurs algorithmes en $O(n \log n)$ ont été proposés pour ce problème classique, utilisant les techniques de balayage ou de diviser pour régner. Nous présentons un algorithme randomisé en temps linéaire,

1. Un polygone est simple si ses segments ne se croisent pas.
2. L'orientation normale suit le sens inverse des aiguilles d'une montre.

c'est-à-dire dont le temps de calcul en espérance est linéaire. D'après nos expériences, il n'est que légèrement plus efficace en pratique que l'algorithme par balayage, mais bien plus facile à implémenter.

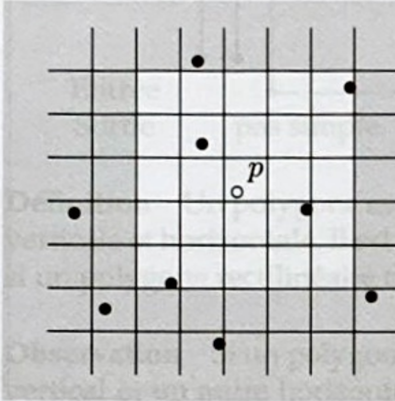


FIGURE 12.3 – Chaque case de la grille contient au plus un point. Au moment de considérer un nouveau point p, il suffit de mesurer sa distance avec des points contenues dans les cases voisines (blanches).

L'idée est qu'à tout moment, on a déjà découvert une paire de points à distance d et on se demande s'il existe une autre paire avec une distance moindre. Pour cela, on divise l'espace en une grille de pas $d/2$ dans les deux directions. Chaque point appartient ainsi à une case de la grille. Soit P l'ensemble des points pour lesquels on a déjà vérifié que les distances entre toute paire de points de P est au moins d. Alors chaque case de la grille contient au plus un point de P.

La grille est représentée par un dictionnaire G qui associe à chaque case le point de P qu'elle contient éventuellement. Au moment d'ajouter un point p à P et à G, il suffit de tester sa distance avec les points q contenus dans les 5×5 cases autour de la case de p, voir figure 12.3. Si une paire de points à distance $d' < d$ a été découverte, on recommence la procédure du début avec une nouvelle grille de pas $d'/2$.

Complexité On suppose que les accès à G se font en temps constant et que le calcul de la case contenant un point donné est constant également. L'argument clé est que si les points donnés en entrée sont traités dans un ordre choisi uniformément au hasard, alors au moment du traitement du i-ème point ($3 \leq i \leq n$), on améliore la distance d avec probabilité $1/(i-1)$. Donc la complexité en espérance est de l'ordre de $\sum_{i=3}^{n} i/(i-1)$, donc linéaire en n.

Détails d'implémentation Pour calculer la case associée à un point (x, y) dans une grille d'un pas donné, il suffit de diviser chaque coordonnée par *pas* et d'arrondir par défaut. Une attention particulière est à porter aux coordonnées négatives, car en Python et dans les autres langages par exemple $\lfloor -1/2 \rfloor$ donnera 0 au lieu du -1 voulu.

Enfin, choisir un pas de grille de $d/2$ au lieu de d garantit la présence d'un seul élément par case, ce qui facilite le traitement.

```
from math import hypot    # hypot(dx, dy) = sqrt(dx * dx + dy * dy)
from random import shuffle


def dist(p, q):
    return hypot(p[0] - q[0], p[1] - q[1])


def floor(x, pas):
    return int(x / pas) - int(x < 0)


def cell(point, pas):
    x, y = point
    return (floor(x, pas), floor(y, pas))


def ameliore(S, d):
    G = {}                 # grille
    for p in S:
        (a, b) = cell(p, d / 2.)
        for a1 in range(a - 2, a + 3):
            for b1 in range(b - 2, b + 3):
                if (a1, b1) in G:
                    q = G[a1, b1]
                    pq = dist(p, q)
                    if pq < d:
                        return (pq, p, q)
        G[a, b] = p
    return None


def closest_points(S):
    shuffle(S)
    assert len(S) >= 2
    p = S[0]
    q = S[1]
    d = dist(p, q)
    while d > 0:
        r = ameliore(S, d)
        if r:
            (d, p, q) = r
        else:
            break
    return (p, q)
```