Aufgabe 1: Laubmaschen

Teilnahme-ID: 70408

Bearbeiter/-in dieser Aufgabe:
Robert Vetter

14. April 2024

Inhaltsverzeichnis

1 Lösungsidee

Grundsätzlich lässt sich das Problem des Hausmeisters mithilfe eines zweidimensionalen quadratischen Feldes modellieren, wobei auf jedem Feld anfänglich gleich viel Laub aufzufinden ist. Anhand der Problemstellung wird schon deutlich, dass sich die angegebene Verteilung der Blätter nicht äquivalent auf den Blasevorgang am Rand und in den Ecken übertragen lässt. Hierbei ist wichtig zu beachten, dass die Gesamtzahl der Blätter auf dem Schulhof immer konstant erhalten bleibt und diese nur innerhalb des Schulhofes umverteilt werden. Des Weiteren modellierte ich die 10%ige Wahrscheinlichkeit eines Blattes, beim Blasevorgang das Feld zu wechseln, als feste Wahrscheinlichkeit und nicht als Zufallszahl. Somit blieb es mir erspart, mehrere Simulationen zu durchlaufen zu müssen. Im folgenden werde ich meine Ergänzungen jeweils an Höfen erläutern, die genau 100 Blätter pro Feld besitzen. Zum Blasevorgang am Rand des Hofes stellte ich folgende Regel auf:

> Nach einem gewöhnlichen Blasevorgang würden sich normalerweise 80% der Blätter des Ausgangsfeldes und 90% der Blätter des Zielfeldes noch auf dem Zielfeld befinden. Bei einer anfänglichen gleichmäßigen Verteilung der Blätter müssten somit auf dem Zielfeld 170 Blätter vorzufinden sein. Bewegt man sich nun am Rand, fehlt eine Seite, auf die 10% der Blätter des Ausgangsfeldes abgegeben werden würde. Daher entschied ich, diese 10% auf das Zielfeld zu übertragen. Somit würde bei anfänglich gleichmäßiger Verteilung folgendes Ergebnis zustande kommen:

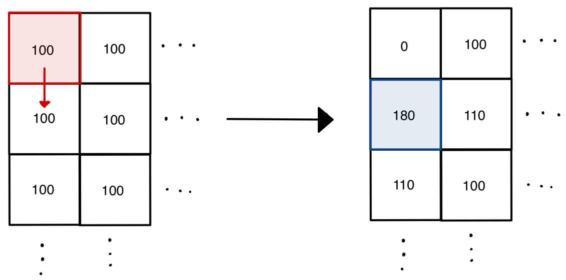

Abbildung 1: Veranschaulichung des Blasevorgangs am Rand

Bei den Ecken wählte ich einen ähnlichen Ansatz. Da hier sozusagen zwei angrenzende Felder außerhalb des Schulhofes liegen, die Blätteranzahl jedoch konstant bleiben muss, lassen sich auf dem Zielfeld nun 190 Blätter auffinden.

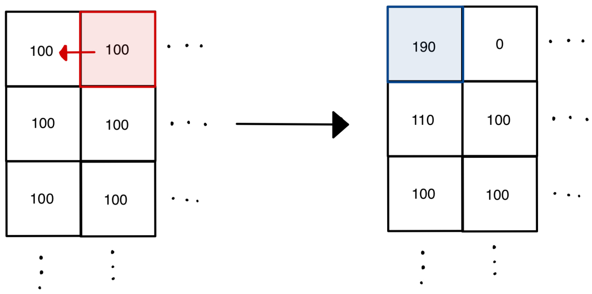

Abbildung 2: Veranschaulichung des Blasevorgangs in den Ecken

Auf der Suche nach einer Strategie, wie der Hausmeister möglich viel Laub auf einem Zielfeld ansammeln kann, gibt es verschiedene Ansätze. Da die Aufgabenstellung nicht explizit nach einem effizienten Algorithmus sucht, könnte man mithilfe von $Brute-Force$ das Problem lösen. Hier wählt man einfach immer zusätzlich einen Blasevorgang aus und tut dies solange, bis man ein zufriedenstellendes Ergebnis erzielt. Diese Methode würde jedoch insbesondere bei größeren Schulhöfen sehr lange dauern. Daher entschied ich mich gegen diese Methode.

Das Problem kann jedoch auch wie folgt formuliert werden: Gegeben sei eine Menge $\mathcal{A}$ von Matrizen und die Vektoren x und y. Die Herausforderung ist, diese Matrizen $A_1, \ldots, A_k \in \mathcal{A}$ zu finden, sodass sie die den Ausdruck $xA_1 \cdots A_k y$ maximieren. Speziell ist hierbei jeder Vektor Element von $\mathbb{R}^{n^2}$ und repräsentiert die Anzahl der Blätter in jedem Rasterfeld. Der Vektor x besteht in jeder Komponente aus der Zahl 100 (anfängliche Blätterzahl auf einem Feld), und y ist der Vektor, der in der Komponente, die dem Zielrasterfeld entspricht, 1 und sonst 0 enthält. Das Blasen des Laubbläsers in eine bestimmte Richtung von einem bestimmten Feld aus wirkt wie eine lineare Abbildung auf diesen Vektoren und kann daher als Matrix dargestellt werden. Somit hat man für jede mögliche Blasrichtung des Laubbläsers eine Matrix in $\mathcal{A}$. Somit könnte man lineare Programmierung als Lösung verwenden. Hierzu definiert man vorerst $x_0 = x$ und $x_i = x_{i-1}A_i$. Anschließend werden ganzzahlige Variablen eingeführt $t_{i,A}$, die die Werte 0 oder 1 annehmen können, für jedes $i \in \{1, \ldots, k\}$ und jedes $A \in \mathcal{A}$, sowie reelle Variablen $w_{i,A}$. Es wird sichergestellt, dass für jedes i genau eine der Variablen $t_{i,A}$ den Wert 1 hat (die anderen 0), sodass

$$A_i = \sum_{A \in \mathcal{A}} t_{i,A} A,$$

und dass

$$w_{i,A} = \begin{cases} x_{i-1}A & \text{falls } t_{i,A} = 1 \\ 0 & \text{falls } t_{i,A} = 0. \end{cases}$$

Diese Bedingungen werden mit den Einschränkungen $0 \leq t_{i,A} \leq 1$,

$$\sum_{A \in \mathcal{A}} t_{i,A} = 1,$$
$$x_i = \sum_{A \in \mathcal{A}} w_{i,A},$$

und

$$
\begin{aligned}
(w_{i,A})_j &\leq (x_{i-1}A)_j \\
(w_{i,A})_j &\leq 100n^2 t_{i,A} \\
(w_{i,A})_j &\geq 0 \\
(w_{i,A})_j &\geq (x_{i-1}A)_j - 100n^2(1 - t_{i,A}).
\end{aligned}
$$

erreicht. Nun kann man $x_k y$ mit einer gebräuchlichen Integer-Linear-Programming-Software maximieren. Obwohl diese Herleitung mathematisch relativ einfach erscheint, ist die Implementierung davon wesentlich herausfordernder. Daher begab ich mich auf die Suche nach einem anderen Algorithmus, der die Laubblätter auf wenigen Haufen ansammelt. Dabei müssen diese erstmal irgendwie komprimiert werden. Daher kam mir die Idee, alle Blätter erstmal in der oberen Reihe zu sammeln. Nachfolgend habe ich dies beispielhaft an einem $6x6$ Schulhof eingezeichnet.

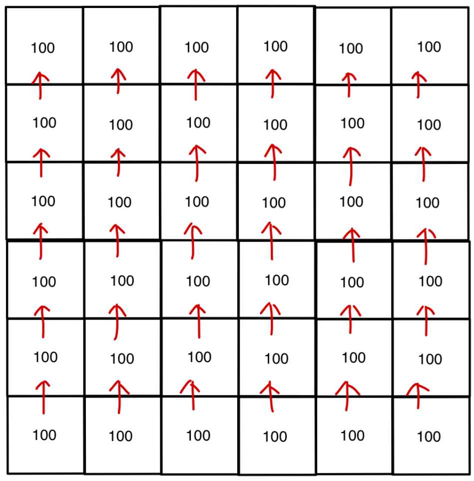

Abbildung 3: Sammeln aller Blätter in Reihe 0

Hat man nun alle Blätter in der obersten Reihe gesammelt (wobei jedes Feld eine ungefähre Blattanzahl von 600 besitzt), müssen sie irgendwie weiter auf wenige Felder konzentriert werden. Dabei ist es möglich, nahezu das gesamte Laub auf ein Rand- bzw. Eckfeld zu konzentrieren. Dieser Mechanismus ist beispielhaft im Folgenden an einem $3x3$ Feld abgebildet:

Abbildung 4: Nahezu verlustloses Übertragen auf ein anderes Randfeld

Bei diesem Feld befinden sich bereits alle Blätter in der obersten Reihe (je 300 pro Feld). Verschiebt man nun vom Feld (0,0) alle Blätter eins nach rechts, dann ergibt sich mit den oben aufgestellten Regeln ein Seitenverlust von 30 Blättern, wobei am Ende 540 Blätter auf dem Zielfeld aufzufinden sind (aufgrund des Verlustes um 10% nach Feld (0, 2)). Bläst man anschließend Feld (1, 1) nach oben, dann hat man sozusagen eine Periode durchlaufen. Nun befinden sich auf dem Ausgangsfeld nur noch $\frac{1}{100}$ der usprünglich vorhandenen Blätter (da zweimal Seitenverluste). Dieses Mechanismus kann man nun immer wieder wiederholen. Die Blätteranzahl nimmt somit gemäß folgender Funktion ab:

$$n(t) = 300 \cdot (\frac{1}{100})^t \tag{1}$$

Entsprechend gilt für das Verhalten gegen unendlich:

$$\lim_{x \to \infty} n(t) = 0 \tag{2}$$

Bei sehr hohen Anzahlen an Perioden, kann somit das verbleibende Laub als 0 angenommen werden. Basierend auf dieser Erkenntnis lässt sich also nun das gesamte Laub auf ein Eck- oder Randfeld konzentrieren. Ich entschied mich hierbei für das Feld (0, 1). Anschließend wird das gesamte Laub ein Feld nach unten bewegt und danach das Feld (1, 2) nach links:

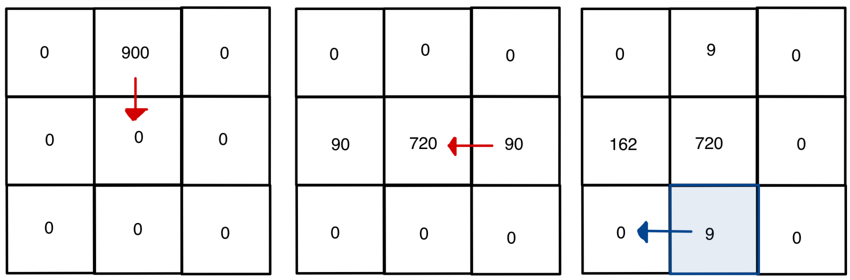

Abbildung 5: Konzentration des Laubes in einer Ecke

Das blau markierte Feld wird nun wieder über mehrfache Iterationen nach links bewegt, solange, bis der Anteil der Blätter auf diesem Feld verschwindend gering ist. Danach kann Feld zweimal (2,0) nach unten bewegt werden, was den Startpunkt des finalen Zyklus markiert. Dieser ist nachfolgend dargestellt:

Abbildung 6: Zyklus zur Verdichtung des Laubes auf einem Feld

Zuerst wird in diesem Feld (0, 1) nach links verschoben. Ascnhließend wird Feld (0, 0) nach oben verschoben, wobei 10% auf das Zielfeld übergehen. Die Verluste auf Feld (2, 0) werden danach nach unten verschoben, wonach Feld (1, 0) wiederum nach unten verschoben wird. Dieser Zyklus kann solange wiederholt werden, bis nahezu das gesamte Laub auf Feld (1, 1) gelagert ist.

2 Umsetzung

Die Umsetzung dieser Aufgabe erfolgte in Python. Dafür ist die Klasse `LeafBlowerSimulation` entworfen worden, um den Prozess des Laubblasens auf einem quadratischen Schulhof zu simulieren. Diese Klasse verwendet ein Gittermodell, das den Schulhof darstellt, wobei jedes Element des Gitters eine bestimmte Menge an Laub enthält. Die Klasse beinhaltet mehrere Methoden:

- `__init__(self, size, initial_leaves)`: Dieser Konstruktor initialisiert das Gitter mit einer gegebenen Größe und einer Anfangsmenge an Laub in jedem Quadrat.
- `blow_leaves(self, moves)`: Diese Methode nimmt eine Liste von Bewegungen entgegen, wobei jede Bewegung eine Position und eine Richtung spezifiziert, und aktualisiert das Gitter entsprechend den Laubblasregeln.
- `_is_valid_move(self, x, y, direction)`: Eine Hilfsmethode, die überprüft, ob eine vorgeschlagene Bewegung innerhalb der Grenzen des Gitters gültig ist.
- `_get_direction_delta(self, direction)`: Eine Methode, die die notwendige Indexänderung für eine gegebene Bewegungsrichtung zurückgibt.
- `_move_leaves(self, x, y, nx, ny)`: Verarbeitet die tatsächliche Verschiebung der Blätter im Gitter, unter Berücksichtigung der Streuverluste und der spezifischen Bewegungslogik (insbesondere Randbedingungen durch if-Abfragen).
- `display_grid(self)`: Zeigt den aktuellen Zustand des Gitters an und berechnet die Gesamtanzahl der Blätter.

Diese Klasse bildet die Grundlage der Simulation. Um die Strategie zu implementieren, schrieb ich eine Methode *determine_best_moves* mit der Größe des Feldes sowie der ursprünglichen Anzahl an Blättern als Parameter. Diese schreibt dann einfach die Anweisungen für den Hausmeister gemäß der in der Lösungsidee vorgestellten Strategie in eine Liste, welche anschließend an die Klasse übergeben wird, die die Simulation ausführt. Nun möchte ich die Zeitkomplexität meiner Klasse analysieren. Die Zeitkomplexität der verschiedenen Methoden in der `LeafBlowerSimulation` kann wie folgt beschrieben werden:

- **Initialisierung (`__init__`):** Die Initialisierung des Gitters der Größe $n \times n$ mit einem Anfangswert erfolgt in $O(n^2)$, da jede Zelle im Gitter individuell gesetzt wird.
- **Blattbewegung (`blow_leaves`):** Diese Methode durchläuft alle Bewegungen im übergebenen Array `moves` und führt entsprechende Aktualisierungen im Gitter durch. Die Komplexität hängt von der Anzahl der Bewegungen m ab und ist $O(m)$.
- **Hilfsmethoden:**
  - `_is_valid_move` und `_get_direction_delta` haben jeweils eine konstante Laufzeit $O(1)$.
  - `_move_leaves` führt Updates in einer begrenzten Anzahl von Gitterzellen durch und hat daher ebenfalls eine konstante Zeitkomplexität $O(1)$.
- **Gitteranzeige (`display_grid`):** Das Drucken des $n \times n$-Gitters erfordert das Durchlaufen aller Zellen, was zu einer Komplexität von $O(n^2)$ führt.

Die Gesamtlaufzeit der Simulation ist daher hauptsächlich abhängig von der Größe des Gitters n und der Anzahl der Bewegungen m, mit einer Gesamtkomplexität von $O(n^2 + m)$. Nun möchte ich mich der Methode *determine_best_moves* widmen:

1. **Sammeln aller Blätter in Reihe 0:**
   - Die äußere Schleife läuft von $n-1$ bis 1, insgesamt $n-1$ Durchläufe.
   - Die innere Schleife läuft von 0 bis $n-1$, was n Durchläufe ergibt.
   - Jede Iteration fügt eine Bewegung hinzu, was zu einer Komplexität von $O(n^2)$ führt.
2. **Platzierung aller Blätter in (0, 1) und entsprechende Bewegungen:**
   - Eine konstante Anzahl von Durchläufen durch die erste Spalte (*hier 1 Durchlauf*).

- Für jede Position werden 50× *initial_leaves* Bewegungen in zwei Richtungen hinzugefügt.
- Komplexität: $O(initial\_leaves)$.

3. **Bewegungen von der oberen Reihe zu bestimmten Positionen (0, n-1 bis 3):**
   - Durchläuft Spalten von $n - 1$ bis 3, was etwa n Durchläufe sind.
   - Für jede Spalte 50× *initial_leaves* Bewegungen.
   - Komplexität: $O(n \times initial\_leaves)$.
4. **Verdichtung in Ecke und periodische Bewegungen:**
   - Fügt eine konstante Anzahl von Bewegungen hinzu, multipliziert mit $50 \times initial\_leaves$.
   - Komplexität: $O(initial\_leaves)$.

Die gesamte Komplexität der Funktion ergibt sich aus der Summe der Komplexitäten aller Schritte und beträgt $O(n^2 + initial\_leaves \times n)$. Dabei dominieren das quadratische Wachstum in Bezug auf die Größe des Gitters n und die lineare Abhängigkeit von der Anzahl der initialen Blätter multipliziert mit n.

3 Beispiele

Ich legte fest, dass der Schulhof mindestens eine Größe von $4x4$ Feldern besitzen muss. Nachfolgend sind die Ausgaben des Programms bei verschiedenen Schulhofgrößen und Laubzahlen dargestellt:

<table>
<tr><th>Finale Laubkonstellation</th><th>n; Anzahl Blätter / Feld</th></tr>
<tr><td>[[0.000 0.000 0.000 0.000]
[0.000 1600.000 0.000 0.000]
[0.000 0.000 0.000 0.000]
[0.000 0.000 0.000 0.000]]
Total leaves in grid: 1600.000
Benötigte Zeit: 0.014148473739624023 Sekunden</td><td>n: 4
Anzahl Blätter / Feld: 100</td></tr>
<tr><td>[[0.000 0.000 0.000 0.000 0.000]
[0.000 2500.000 0.000 0.000 0.000]
[0.000 0.000 0.000 0.000 0.000]
[0.000 0.000 0.000 0.000 0.000]
[0.000 0.000 0.000 0.000 0.000]]
Total leaves in grid: 2500.000
Benötigte Zeit: 0.011999130249023438 Sekunden</td><td>n: 5
Anzahl Blätter / Feld: 100</td></tr>
<tr><td>[[0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000]
[0.000 5000.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000]
[0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000]
[0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000]
[0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000]
[0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000]
[0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000]
[0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000]
[0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000]
[0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000]]
Total leaves in grid: 5000.000
Benötigte Zeit: 0.01199650764465332 Sekunden</td><td>n: 10
Anzahl Blätter / Feld: 50</td></tr>
<tr><td>[[0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000]
[0.000 10000.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000]
[0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000]
[0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000]
[0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000]
[0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000]
[0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000]
[0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000]
[0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000]
[0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000]]
Total leaves in grid: 10000.000
Benötigte Zeit: 0.030419111251831055 Sekunden</td><td>n: 10
Anzahl Blätter / Feld: 100</td></tr>
<tr><td>[[0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000]
[0.000 20000.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000]
[0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000]
[0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000]
[0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000]
[0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000]
[0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000]
[0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000]
[0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000]
[0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000]]
Total leaves in grid: 20000.000
Benötigte Zeit: 0.07688379287719727 Sekunden</td><td>n: 10
Anzahl Blätter / Feld: 200</td></tr>
<tr><td>Ausgabe zu groß</td><td>n: 100
Anzahl Blätter / Feld: 100
Laufzeit des Programms: 0,2 Sekunden</td></tr>
</table>

4 Quellcode

```
    import numpy as np
import time

class LeafBlowerSimulation:
    def __init__(self, size, initial_leaves):
        self.grid = np.full((size, size), float(initial_leaves), dtype=float)
        self.size = size

    # aktualisiert Gitter, koordiniert Klasse
    def blow_leaves(self, moves):
        for move in moves:
            position, direction = move
            x, y = position
            if not self._is_valid_move(x, y, direction):
                print(f"Warning: Cannot move {direction} from ({x}, {y}) - out of bounds!")
                continue
            dx, dy = self._get_direction_delta(direction)
            nx, ny = x + dx, y + dy
            self._move_leaves(x, y, nx, ny)
        self.display_grid()

    # ueberprueft, ob Bewegung innerhalb des Schulhofes erfolgt
    def _is_valid_move(self, x, y, direction):
        dx, dy = self._get_direction_delta(direction)
        nx, ny = x + dx, y + dy
        return 0 <= nx < self.size and 0 <= ny < self.size

    # moegliche Bewegungsrichtungen
    def _get_direction_delta(self, direction):
        directions = {
            'up': (-1, 0),
            'down': (1, 0),
            'left': (0, -1),
            'right': (0, 1)
        }
        return directions.get(direction, (0, 0))

    # bewegt die Blaetter und berechnet die Verteilungen
    def _move_leaves(self, x, y, nx, ny):
        # grundlegende Infos, die in Aufgabenstellung gegeben waren
        leaves_to_move = 0.8 * self.grid[x, y]
        left_leaves = 0.1 * self.grid[x, y]
        right_leaves = 0.1 * self.grid[x, y]
        push_leaves = 0.1 * self.grid[nx, ny]

        # Blaetter auf einzelnen Feldern werden berechnet
        self.grid[x, y] -= leaves_to_move + left_leaves + right_leaves
        self.grid[nx, ny] += leaves_to_move - push_leaves

        # Berechnung der Koordinaten der betroffenen Felder
        dx, dy = nx - x, ny - y
        if dx != 0:
            lateral_left = (nx, ny-1)
            lateral_right = (nx, ny+1)
            front = (nx + dx, ny)
        else:
            lateral_left = (nx-1, ny)
            lateral_right = (nx+1, ny)
            front = (nx, ny + dy)

        # Blaetterverteilung an Raendern wird berechnet (eigene Regeln, siehe Loesungsidee)
        if 0 <= lateral_left[1] < self.size and 0 <= lateral_left[0] < self.size:
            self.grid[lateral_left] += left_leaves
        else:
            self.grid[nx, ny] += left_leaves
        if 0 <= lateral_right[1] < self.size and 0 <= lateral_right[0] < self.size:
            self.grid[lateral_right] += right_leaves
        else:
            self.grid[nx, ny] += right_leaves
        if 0 <= front[0] < self.size and 0 <= front[1] < self.size:
            self.grid[front] += push_leaves
```

```
        else:
            self.grid[nx, ny] += push_leaves
        # Nullify nearly zero values
        self.grid[np.abs(self.grid) < 1e-6] = 0.0

    # lediglich das Gitter anzeigen
    def display_grid(self):
        column_width = 10

        formatter = lambda x: f"{x: {column_width}.3f}"

        np.set_printoptions(precision=3, suppress=True, linewidth=200,
        formatter={'float_kind': formatter})
        print(self.grid)
        total_leaves = np.sum(self.grid)
        print(f"Total leaves in grid: {total_leaves:.3f}")

# Loesungsstrategie (ausfuehrliche Erklaerung in Loesungsidee)
def determine_best_moves(size, initial_leaves):
    moves = []

    # Sammeln aller Blaetter in Reihe 0
    for i in range(size-1, 0, -1):
        for j in range(size):
            moves.append([(i, j), 'up'])

    # Platzierung aller Blaetter in (0, 1)
    for spalte in range(1):
        for _ in range(50 * initial_leaves):
            moves += [[(0, spalte), 'right']]
            moves += [[(1, spalte + 1), 'up']]

    for spalte in range(size-1, 2, -1):
        for _ in range(50 * initial_leaves):
            moves += [[(0, spalte), 'left']]
            moves += [[(1, spalte - 1), 'up']]

    # Verdichtung in Ecke
    moves += [[(0, 1), 'down']]
    moves += [[(1, 2), 'left']]

    for _ in range(50 * initial_leaves):
        moves += [[(2, 1), 'left']]
        moves += [[(3, 0), 'up']]
    moves += [[(1, 0), 'up']]


    # Periodisches Abgeben der 10% zur Seite
    for _ in range(50 * initial_leaves):
        moves += [[(0, 1), 'left']]
        moves += [[(0, 0), 'down']]
        moves += [[(2, 0), 'up']]
        moves += [[(1, 0), 'up']]

    return moves
n = 100
initial_leaves = 100
if n == 0 or initial_leaves == 0:
    print("Stopp! Es gibt nichts zu kehren!")
if n > 3:
    sim = LeafBlowerSimulation(size=n, initial_leaves=initial_leaves)
    print("Initial grid:")
    sim.display_grid()
    start = time.time()
    # best_moves = [[(0, 1), 'left'], [(0, 4), 'down']]
    best_moves = determine_best_moves(n, initial_leaves)
    end = time.time()
    sim.blow_leaves(best_moves)
    print("Benoetigte Zeit: ", end - start, " Sekunden")
else:
    print("Mindestgroesse vom Schulhof sind 4x4 Felder!")
```