

Aufgabe 1: Laubmaschen

Teilnahme-ID: 69408

Bearbeiter dieser Aufgabe:
Tim Krome

12. April 2024

Lösungsidee	2
Wirkung eines Blasvorgangs.....	3
Modellierung der Wahrscheinlichkeiten.....	7
Formulierung als Bernoulli-Experiment	7
Berechnen von $P(X=k)$	9
Möglichkeiten zur Modellierung der Regeln	10
Berechnen des Resultats eines Blasvorgangs (Pseudocode)	12
Trivialer Ansatz: Heuristisches Vorgehen (1. Ansatz)	13
Kenngrößen zur Bewertung des Zustands eines Hofs	14
Kombination der Heuristiken	19
Behandeln des Rands	20
Gesamt-Algorithmus	24
Laufzeit	25
Kritik am Algorithmus	25
Besseres Vorgehen: Generalisierte Ablaufpläne (2. Ansatz)	26
Ziel	26
Auf welchen Höfen kann das Laub nicht auf Q geblasen werden?	27
Zugrundeliegende Strategie.....	27
Definitionen: Generalisierte Ablaufpläne und Muster.....	29
Können die Rand- und Eckfelder vollständig geleert werden?	30
Muster zum Leeren von Rand- und Eckfeldern	31
Bauen des generalisierten Ablaufplans.....	32
1. Phase: Unterste Reihe des Hofs befreien	33
2. Phase: Laub auf oberste Reihe bringen.....	34
3. Phase: Laub auf der obersten Reihe konzentrieren.....	35
4. Phase: Gesamtes Laub auf Feld Q verschieben	36
Optimierungen	41
Gesamt-Algorithmus und Kritik	41
Laufzeit	42
Kombination der Verfahren (3. Ansatz)	43
Parametrisierung	43
Zusammenfassung.....	43
Bereits in Solver 1 bis 3 enthaltene Erweiterungen.....	43
Erweiterung 1: Keine Mauer-Umrundung des Hofs.....	44
Laufzeit	47
Erweiterung 2: Mehrere Laubbläser.....	47
Synergien ausnutzen.....	49
Laufzeit	50
j Laubbläser.....	51
Erweiterung 3: Verschiedene Laubtypen.....	51
Weitere Erweiterungsideen (mit Lösungsansätzen)	51
Andere Hofformen: Rechtwinklige Polygone	51
Weitere Ideen	53
Umsetzung	53

Hilfsfunktionen zur Berechnung von Binomialverteilungen (binomial_util.py).....	54
Definieren einer Rules-Klasse (hof.py).....	55
Definieren einer Hof-Klasse (hof.py).....	56
Grundaufgabe Ansatz 1	57
Grundaufgabe Ansatz 2:	59
Beispiele – Grundaufgabe.....	60
Ansatz 1 (aufgabe1_solver.py).....	61
Beispiel 1.....	61
Beispiel 2.....	62
Beispiel 3.....	63
Beispiel 4.....	64
Beispiel 5.....	65
Beispiel 6.....	66
Ansatz 2 (aufgabe2_solver.py).....	68
Beispiel 1.....	68
Beispiel 2.....	71
Beispiel 3.....	72
Beispiel 4.....	73
Beispiel 5.....	75
Beispiel 6.....	76
Beispiel 7.....	78
Ansatz 3 (aufgabe3_solver.py).....	80
Beispiel 1.....	80
Beispiel 2.....	81
Beispiel 3.....	82
Beispiele – Erweiterung 1: Keine Mauer-Umrandung.....	83
Beispiel 1	84
Beispiel 2	85
Beispiele – Erweiterung 2: Zwei Laubbläser	86
Beispiel 1	87
Beispiel 2	88
Beispiele – Erweiterung 3: Verschiedene Laubtypen.....	90
Beispiel 1	90
Beispiel 2	91
Quellcode.....	92
Auszüge aus binomial_util.py	92
Auszüge aus hof.py	94
Grundaufgabe: Ansatz 1	97
Grundaufgabe: Ansatz 2	102
Grundaufgabe: Ansatz 3	110
Erweiterung 1 (basierend auf Ansatz 2)	110
Erweiterung 2 (basierend auf Ansatz 2)	115

Lösungsidee

Der Schulhof besteht aus gleich großen Planquadraten, die in einem gleichmäßigen Raster angeordnet sind. Der Schulhof selbst ist dabei quadratisch und hat somit die Maße (n Planquadrate, n Planquadrate) mit $n \in \mathbb{N}$. Der Aufgabe nach ist das Laub anfangs gleichmäßig auf dem Hof verteilt, d.h. auf jedem Planquadrat befinden sich gleich viele Blätter.

Auf dem Hof führt der Hausmeister Blasoperationen durch. Eine Blasoperation kann als Funktion verstanden werden, die von zwei Parametern abhängig ist: Das Feld, auf dem der Hausmeister steht (im Folgenden als „Feld 0“ bezeichnet), und die Richtung, in die er bläst. Im Folgenden verwende ich diese Kurzschreibweise für eine Blasoperation, die vom Feld 0 *feld0* aus in Richtung *direction* bläst: *blase(feld, direction)*.

Bei Anwendung der Funktion ändern sich die Blattanzahlen auf den einzelnen Feldern. Die Modellierung einer Blasoperation wird dadurch erschwert, dass diese **Änderungen vom Zufall abhängen**. Die in der Aufgabe definierten Blasregeln sind probabilistisch, sie geben für jedes Blatt auf Feld A und Feld B die Wahrscheinlichkeit an, dass dieses Blatt auf ein bestimmtes anderes Feld geweht wird. Feld A befindet sich hierbei unmittelbar vor der Luftaustrittsdüse des Laubbläzers, Feld B befindet sich hinter Feld A (siehe Abbildung 1).

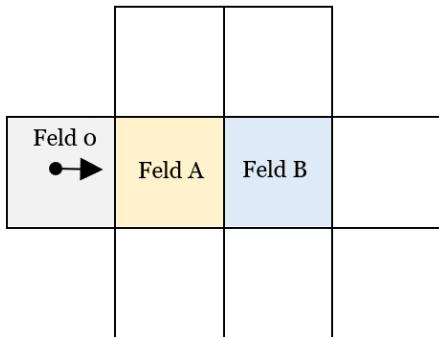


Abbildung 1: Feld A und B in Relation zu Hausmeister (Punkt) und Blasrichtung (Pfeil)

Der Hof kann als zweidimensionale Matrix modelliert werden. Jeder Eintrag der Matrix korrespondiert mit einem Feld des Hofs und speichert die Anzahl an Blättern, die sich auf dem Feld befinden. Die Modellierung erfolgt so, dass das Feld in der Zeile x und Spalte y durch den Matrix-Eintrag am Index (x,y) entspricht.

(0,0)	(1,0)	(2,0)	(3,0)
(0,1)			
(0,2)			

oben ↑

rechts →

Abbildung 2: Veranschaulichung der Felder-Indizierung (jedes Feld ist mit seinem Index in der Matrix beschrieben)

Zur Darstellung einer Blasrichtung wird ein Einheitsvektor verwendet, der in die Richtung zeigt, in die der Laubbläser gehalten wird:

- Richtung $(1,0) \triangleq$ rechts
- Richtung $(-1,0) \triangleq$ links
- Richtung $(0,1) \triangleq$ oben
- Richtung $(0,-1) \triangleq$ unten

Wirkung eines Blasvorgangs

Die Aufgabe schreibt für die Wirkung eines Blasvorgangs folgende Regeln vor:

Vorher:	Nachher:	
		$P_{A_to_B} = 0,8$ $P_{A_to_above_B} = 0,1$ $P_{A_to_below_B} = 0,1$ $P_{B_stay_on_B} = 0,9$ $P_{B_to_beyond_B} = 0,1$

Diese Regeln sind nur anwendbar, wenn Feld A und B sowie die Nachbarfelder (horizontal und vertikal, nicht diagonal) von Feld B tatsächlich innerhalb des Hofs existieren. Dem ist nur der Fall, wenn Feld 0, Feld A und Feld B keine Rand- oder Eckfelder sind. Für alle anderen möglichen Fälle müssen zusätzliche Regeln definiert werden. Hierbei habe ich folgende Regeln berücksichtigt (die Reihenfolge gibt die Priorisierung der Prinzipien an, das erste Prinzip wird dabei am höchsten priorisiert):

1. In der Aufgabe werden keine Aussagen zur Begrenzung des Schulhofs getroffen, wodurch man einiges an Freiraum bei der Definition der zusätzlichen Blasregeln hat.

Ich lege zunächst fest, dass der Hof von einer hohen Mauer umgeben ist, über die keine Blätter fliegen können. Dies hat zur Folge, dass **die Gesamtbläumenge auf dem Hof konstant bleiben muss (1. Prinzip)**.

Anmerkung: Zunächst erscheint es logisch, keine Operationen zuzulassen, die zu einer Verringerung der Gesamtbläumenge auf dem Hof führen: Sonst könnte man ja einfach alle Blätter vom Hof runterblasen und auf die Art den Hof von Blättern befreien, was den Sinn der Aufgabe sicher nicht treffen würde.

Bei genauerer Betrachtung der Aufgabe stellt man jedoch fest, dass das Ziel des Hausmeisters klar definiert ist *als möglichst viele Blätter auf Feld Q bringen* - und nicht *den Hof reinigen*. Vom Hof heruntergeblasene Blätter würden sich also negativ auf das Erreichen des Ziels auswirken, da diese das System verlassen haben und somit nicht mehr auf Q geblasen werden können. Würde man doch erlauben, dass Blätter den Hof durch Seitenabtriebseffekte verlassen können, dann wäre die Entwicklung einer Strategie tatsächlich schwieriger. (Mehr dazu in *Lösungsidee > Erweiterung 1*)

2. Die ergänzten Regeln sollen sich **an den von der Aufgabe vorgeschriebenen Regeln orientieren und nur dort, wo es notwendig ist** (weil beteiligte Felder außerhalb vom Hof liegen) **von ihnen abweichen (2. Prinzip)**.
3. Dort, wo von den vorgeschriebenen Regeln abgewichen werden muss, sollen die neuen Regeln **realistisch** sein und sich **an tatsächlichen physikalischen Gesetzmäßigkeiten orientieren (3. Prinzip)**.

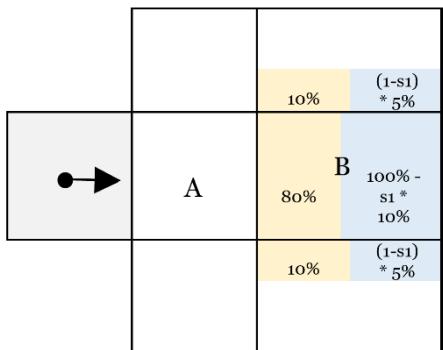
Es ist allerdings nicht möglich, allein aus den gegebenen Informationen mithilfe physikalischer Gesetze die exakten Wahrscheinlichkeiten für jeden Sonderfall zu ermitteln (es fehlen

wesentliche Informationen wie die Oberflächenbeschaffenheit der Wand, die Reibung vorbeifliegender Blätter beeinflusst, es ist außerdem in den von der Aufgabe gegebenen Regeln nicht definiert wie die Blätter nach einer Blasoperation innerhalb der Planquadrate verteilt sind, erschwerend kommt die nichtlineare Flugbahn eines Blattes dazu). Bei den in der Aufgabe gegebenen Regeln handelt es sich ja auch nur um ein Modell.

Stattdessen werden plausible Abschätzungen betroffen, basierend auf denen modellhafte Regeln festgelegt werden. Dies erfolgt an vielen Stellen vereinfachend. Zur Generalisierung der Regeln werden außerdem verschiedene Parameter eingeführt. Über diesen Parameter kann die Wahrscheinlichkeit dort festgelegt werden, wo basierend auf den Informationen aus der Aufgabe eine eindeutige Festlegung der Wahrscheinlichkeit nicht möglich ist. Bei einer konkreten Anwendung des entwickelten Programms könnten diese Parameter kalibriert werden, um die Regeln an die tatsächlichen Gegebenheiten anzupassen.

Bei den nachfolgenden Regeldefinitionen ist der Anfangs- bzw. „Vorher“-Zustand immer, dass sich sowohl auf A als auch auf B (falls vorhanden) 100% der Blätter befinden. Dicke Kanten bedeuten, dass sich hier die den Hof eingrenzenden Mauern befinden.

Sonderfall 1: Eine Mauer ist rechts von B



$$P_{A_to_B} = 0,8$$

$$p_{A_to_above_B} = 0,1$$

$$p_{A_to_below_B} = 0,1$$

$$p_{B_stay_on_B} = 1 - 0,1 * s1$$

$$p_{B_to_above_B} = 0,05 * (1-s1)$$

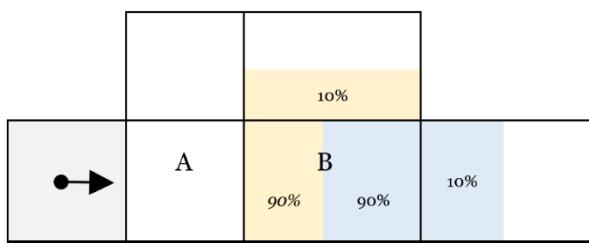
$$p_{B_to_below_B} = 0,05 * (1-s1)$$

Es fliegt kein Laub über die Mauer hinaus (Prinzip 1 erfüllt) und nur der Abtrieb von Feld B (= die 10% des Laubs von Feld B, die sonst auf das Feld rechts von B geflogen wären), fliegen den neuen Regeln nach anders (Prinzip 2 erfüllt).

B, die sonst auf das Feld rechts von B geflogen wären), fliegen den neuen Regeln nach anders (Prinzip 2 erfüllt).

Praktische Beobachtungen zeigen: Wenn Blätter gegen eine Wand geblasen werden, dann fliegen sie bei ausreichender Krafteinwirkung meist zur Seite weg. Mangels Informationen zu den Gegebenheiten lässt sich wie zuvor erläutert nicht eindeutig bestimmen, wie groß der Anteil der Blätter ist, der an der Wand hängen bleibt bzw. nicht auf ein seitliches Feld geweht wird.

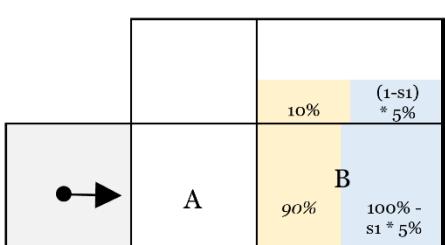
Daher wird der Parameter $s1$ eingeführt. $s1$ gibt den Anteil am regulären Abtrieb von Feld B an, der (bei häufiger Experimentdurchführung durchschnittlich) in Feld B verbleibt und nicht seitlich wegfliegt. Im Folgenden wird mit $s1 = 0,9$ gearbeitet (d.h. ca. 90% des regulären Abtriebs von Feld B verbleibt in B und fliegt nicht zur Seite weg). Dies ist plausibel, da die vom Laubbläser ausgehende Krafteinwirkung am rechten Rand von Feld B nicht mehr besonders stark ist (das ist daran erkennbar, dass im Normalfall nur ca. 10% der Blätter von B abgetrieben werden). Prinzip 3 wurde also auch berücksichtigt.

Sonderfall 2: Eine Mauer ist unter B

$$\begin{aligned} P_{A_to_B} &= 0,9 \\ P_{A_to_above_B} &= 0,1 \\ P_{B_stay_on_B} &= 0,9 \\ P_{B_to_beyond_B} &= 0,1 \end{aligned}$$

Es wird vereinfachend davon ausgegangen, dass der Seitenabtrieb von Feld A, der regulär auf das Feld unter B geflogen wäre, jetzt stattdessen vollständig auf Feld B geweht wird.

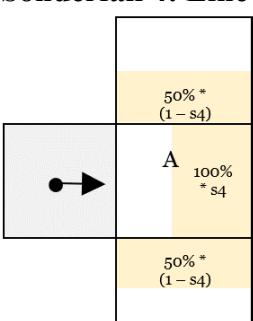
Dieselben Regeln gelten analog, wenn sich die horizontale Mauer über Feld A, B und 0 befindet (in diesem Fall sind die Regeln an der Achse AB zu spiegeln).

Sonderfall 3: B ist ein Eckfeld

$$\begin{aligned} p_{A_to_B} &= 0,9 \\ p_{A_to_above_B} &= 0,1 \\ p_{B_stay_on_B} &= 1 - 0,05 * (1-s1) \\ p_{B_to_above_B} &= 0,05 * s1 \end{aligned}$$

Diese Regeln ergeben sich aus Kombination von Sonderfall 1 und 2, weshalb der Parameter $s1$ von Sonderfall 1 übernommen wird. Das Laub, das in Sonderfall 1 von Feld B auf das Feld unter B geweht worden wäre, wird jetzt in die Mauerecke Ecke von Feld B geweht.

Dieselben Regeln gelten analog, wenn sich die horizontale Mauer über Feld B befindet.

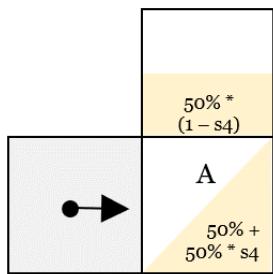
Sonderfall 4: Eine Mauer ist rechts von A (→ B existiert nicht)

$$\begin{aligned} p_{A_to_above_A} &= 0,5 * (1 - s4) \\ p_{A_to_below_A} &= 0,5 * (1 - s4) \\ p_{A_stay_on_A} &= 1 * s4 \end{aligned}$$

Den regulären Regeln nach ist der Seitenabtrieb des Laubs von Feld A oben und unten gleich groß. Daher gehe ich davon aus, dass dem auch in diesem Sonderfall so ist.

Es wird der Parameter $s4$ festgelegt, der festlegt, wie groß der Anteil des auf Feld A verbleibenden Laubs ist. Wegen der starken Krafteinwirkung des Laubbläzers auf Feld A (erkennbar daran, dass dieses Feld im Normalfall vollständig geleert wird) ist es plausibel, dass auch in diesem Sonderfall fast das ganze Laub von Feld A heruntergeweht wird. Es wird daher im Folgenden mit $s4 = 0,05$ gearbeitet.

Sonderfall 5: A ist ein Eckfeld (→ B existiert nicht)

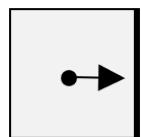


$$p_{A_to_above_A} = 0,5 * (1 - s4)$$

$$p_{A_stay_on_A} = 0,5 + 0,5 * s4$$

Das Laub, das normalerweise auf das Feld unter A geweht worden wäre, wird jetzt in die Mauerecke geblasen.

Sonderfall 6: Feld 0 ist ein Rand- oder Eckfeld



Es wird davon ausgegangen, dass der Luftstrom an der Mauer nicht stark genug reflektiert wird, um die Blätter nennenswert zu bewegen. Daher wird festgelegt, dass in diesem Fall nichts passiert. Das direkte Blasen gegen die Mauer ist somit sinnlos. Im Übrigen wäre es auch nicht förderlich für den Zustand der Mauer. Es kann nicht im Sinne des Hausmeisters sein, den Schulhof zu demolieren. Daher wird in der Strategie auf ein direktes Blasen gegen die Mauer verzichtet.

Modellierung der Wahrscheinlichkeiten

Angenommen auf Feld A befindet sich *genau ein Blatt*. Wird nun mit einem Laubbläser von Feld 0 aus in Richtung A geblasen, dann handelt es sich hierbei um ein Zufallsexperiment mit drei möglichen Ereignissen:

Experiment: Das Blatt auf Feld A wird geblasen		
Ereignis 1: Blatt landet auf B (im folgenden E_1) $P(E_1) = p_{A_to_B}$ (den Blasregeln entnehmbar)	Ereignis 2: Blatt landet auf dem Feld über B (E_2) $P(E_2) = p_{A_to_above_B}$	Ereignis 3: Blatt landet auf dem Feld unter B (E_3) $P(E_3) = p_{A_to_below_B}$

Formulierung als Bernoulli-Experiment

Wie bei jedem Zufallsexperiment, das eine endliche Anzahl an Ereignissen hat, lässt sich auch dieses Zufallsexperiment in mehrere miteinander verknüpften Bernoulli-Experimenten umformulieren. Ein Bernoulli-Experiment hat nur ein Ereignis, das entweder eintritt oder nicht eintritt. Das obige Zufallsexperiment lässt sich in zwei miteinander verknüpfte Bernoulli-Experimente umformulieren¹:

Das erste Bernoulli-Experiment überprüft, ob das Ereignis „Blatt landet auf Feld B“ (E_1) zutrifft – die Wahrscheinlichkeit hierfür ist $p_{A_to_B}$ und in den Regeln für alle Fälle definiert. Trifft (E_1) nicht zu, dann wird im zweiten Bernoulli-Experiment (E_2) überprüft. Da (E_2) nur dann angewandt wird, wenn (E_1) nicht zutrifft, handelt es sich bei der Wahrscheinlichkeit dafür, dass das zweite Bernoulli-Experiment wahr ist, um die bedingte

¹ Allgemein lässt sich mit dem im Folgenden beschriebenen Vorgehen ein Zufallsexperiment mit n möglichen Ereignissen in $n-1$ miteinander verknüpfte Bernoulli-Experimente umformen.

Wahrscheinlichkeit $P_{-E_1}(E_2) = P(\neg E_1 \cap E_2) / P(\neg E_1) =^2 P(E_2) / P(\neg E_1) = (p_{A_to_above_B}) / (1 - p_{A_to_B})$.

Ereignis von Bernoulli-Experiment 1: Das Blatt von A landet auf B (E_1)

Wahr $p = P(E_1) = p_{A_to_B}$	Falsch $p = 1 - P(E_1) = P(\neg E_1) = 1 - p_{A_to_B}$
Ereignis von Bernoulli-Experiment 2: Das Blatt von A landet auf dem Feld über B (E_2)	
Wahr $p = P_{-E_1}(E_2) = P(E_2) / P(\neg E_1) = (p_{A_to_above_B}) / (1 - p_{A_to_B})$	Falsch $p = 1 - P_{-E_1}(E_2) = 1 - (p_{A_to_above_B}) / (1 - p_{A_to_B})$ Der einzige übrig bleibende Fall ist (E_3), der folglich auf das Blatt zutreffen muss.

Angenommen, es befinden sich nicht mehr nur 1 Blatt, sondern x Blätter auf Feld A (mit $x \in \mathbb{N}$), dann lässt sich *Bernoulli-Experiment 1* auf jedes dieser x Blätter einzeln anwenden. Hierdurch erhält man eine Bernoulli-Kette³ der Länge x . Die Voraussetzungen für das Vorhandensein einer Bernoulli-Kette sind:

- *Zwei mögliche Ergebnisse:*
Diese Voraussetzung ist durch die Formulierung als Bernoulli-Experiment erfüllt.
- *Unabhängigkeit der Versuche:*
Es wird vereinfachend davon ausgegangen, dass sich die Blätter beim Fliegen nicht gegenseitig beeinflussen. Dadurch ist diese Voraussetzung auch erfüllt.
- *Konstante Erfolgswahrscheinlichkeit:*
Die Regeln schreiben Wahrscheinlichkeiten fest, die konstant sind bzw. während einem Blasvorgang konstant bleiben.
- *Diskrete Anzahl von Versuchen (n):*
Auf jedem Feld liegt eine diskrete, ganzzahlige Anzahl an Blättern, die die Anzahl an Versuchen bzw. Länge der Bernoulli-Kette darstellt (weil das Bernoulli-Experiment auf jedes Blatt einzeln angewandt wird und somit jedes Blatt als Versuch betrachtet werden kann).

Da die Voraussetzungen erfüllt sind, ist die Modellierung als Bernoulli-Kette legitim.

Um zu bestimmen, wie groß die Wahrscheinlichkeit ist, dass genau k Blätter von Feld A (aus den x Blättern, die auf Feld A liegen) auf Feld B landen, lässt sich folglich die Binomialverteilung verwenden: (hierin besteht der große Vorteil in der Darstellung als Bernoulli-Experiment)

Die Zufallsgröße X zählt, wie viele Blätter von Feld A auf Feld B fliegen. Die Wahrscheinlichkeit, dass k Blätter von Feld A auf Feld B fliegen, beträgt somit $P(X=k)$ mit $n=x$ und $p=P(E_1)$.

Auf die $x-k$ Blätter, die nicht auf Feld B fliegen, lässt sich nun das zweite Bernoulli-Experiment

² $P(\neg E_1 \cap E_2) = P(E_2)$, da auf alle Blätter, auf die E_2 (Blatt liegt auf dem Feld über B) zutrifft, E_1 (Blatt liegt auf B) automatisch nicht zutrifft – das Blatt kann (dem Modell nach) schließlich nicht auf beiden Feldern gleichzeitig liegen.

³ Die Modellierung als Bernoulli-Kette ist zulässig unter der Annahme, dass sich die Blätter in ihrer Flugbahn nicht gegenseitig beeinflussen und es sich somit um voneinander unabhängige Bernoulli-Experimente handelt.

anwenden: Die Zufallsgröße Y zählt, wie viele der Blätter, die nicht auf Feld B fliegen, stattdessen auf dem Feld über B landen. Die Wahrscheinlichkeit, dass j von diesen Blättern auf dem Feld über B landen, beträgt somit $P(Y=j)$ mit $n=x-k$ und $p=P_{\neg E_1}(E_2)$, der Wahrscheinlichkeit des 2. Bernoulli-Experiments.

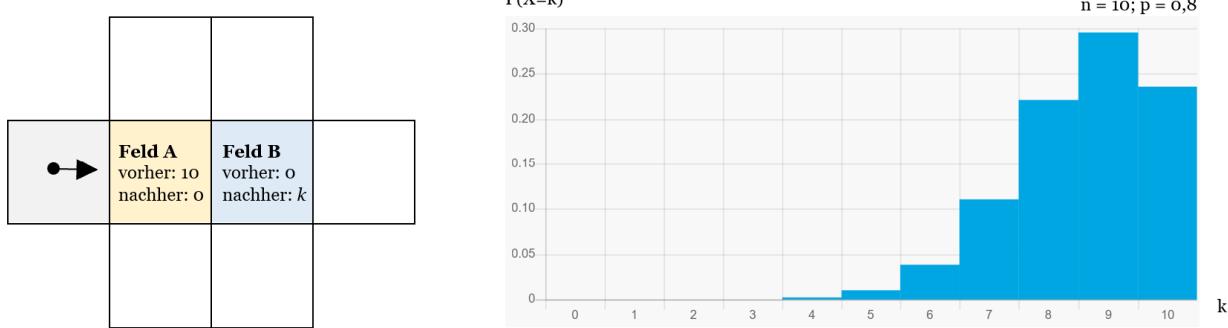


Abbildung 3: Beispiel zur Veranschaulichung – Szenario: 10 Blätter liegen auf Feld A (links). Aus dem Histogramm (rechts) kann entnommen werden, wie groß die Wahrscheinlichkeit dafür ist, dass genau k aus diesen 10 Blättern auf Feld B fliegen.

Nach demselben Prinzip kann auch für ein Blatt, das sich auf Feld B befindet, ein Bernoulli-Experiment definiert werden:

Ereignis von Bernoulli-Experiment 3: Das Blatt von B bleibt auf B

Wahr	Falsch
$p = p_{B_stay_on_B}$	$p = 1 - p_{B_stay_on_B}$

In Sonderfällen (siehe Vorkapitel) müssen die Bernoulli-Experimente entsprechend neu festgelegt werden. Die Festlegung erfolgt dabei nach demselben Vorgehen wie zuvor beschrieben und wird hier nicht im Einzelnen durchgegangen.

Berechnen von $P(X=k)$

Die Wahrscheinlichkeitsfunktion der Binomialverteilung: $P(X=k) = \binom{n}{k} * p^k * (1-p)^{n-k}$ mit n = Länge der Bernoulli-Kette und k = Anzahl Versuche, bei denen das Ereignis des Bernoulli-Experiments eintritt. Eine Implementierung der Funktion in dieser Form zieht jedoch einige Probleme nach sich:

Der Binomialkoeffizient $\binom{n}{k}$ wird **bei großen n-Werten und k-Werten nahe an $\frac{n}{2}$ sehr groß**. So beträgt z.B. $\binom{10}{5}$ nur 252, während $\binom{10}{5}$ bereits 184756 ergibt und bei der Berechnung von $\binom{100}{50}$ eine 100-stellige Zahl herauskommt. Dies liegt daran, dass der Wert von $\binom{n}{k}$ exponentiell wächst, wenn n ansteigt. $\binom{n}{k}$ ist immer positiv und ganzzahlig und kann daher in einem Integer gespeichert werden. Die verwendete Programmiersprache (Python) verwendet für Integer-Werte eine dynamische Speicherverwaltung, was bedeutet, dass Integer-Werte so groß werden können, wie der verfügbare Speicherplatz auf dem System es zulässt – große $\binom{n}{k}$ -Werte sind daher grundsätzlich kein Problem. Problematisch ist allerdings die Multiplikation $\binom{n}{k} * p^k$: Da p^k als Float verliegt,

muss für die Multiplikation auch $\binom{n}{k}$ in einen Float umgewandelt werden. In Python führt die **Konvertierung von langen Integers zu einem OverflowError**. Wie groß der Wert ist, ab dem der *OverflowError* auftritt ist von der Computerarchitektur abhängig, bei 64-bit-Geräten beträgt er ca. 1.7976931348623157e+308. Für große $n > 1000$ kann $\binom{n}{k}$ diesen Wert überschreiten (so ergibt z.B. $\binom{1100}{550}$ eine 329-stellige Zahl).

Außerdem ergibt $p^k * (1 - p)^{n-k}$ für große n verschwinden kleine Dezimalzahlen nahe Null. In Python werden **FLOATS mit einer begrenzten Genauigkeit gespeichert** (15 bis 16 Dezimalstellen werden genau gespeichert), was bei großen n zu einem Genauigkeitsverlust führen würde.

Daher verwende ich zur Berechnung von $P(X=k)$ die **logarithmische Darstellung** der Formel von Bernoulli: $P(X = k) = e^{\ln\left(\binom{n}{k}\right) + k \cdot \ln(p) + (n-k) \cdot \ln(1-p)}$.

Die logarithmische Darstellung ist äquivalent zur üblichen Darstellung. Herleitung:

$$P(X=k) = \binom{n}{k} * p^k * (1 - p)^{n-k} \quad (\text{die übliche Darstellung der Formel von Bernoulli})$$

$$\Leftrightarrow \ln(P(X=k)) = \ln\left(\binom{n}{k} * p^k * (1 - p)^{n-k}\right) \mid \text{Beide Seiten logarithmieren}$$

$$\Leftrightarrow \ln(P(X=k)) = \ln\left(\binom{n}{k}\right) + \ln(p^k) + \ln((1 - p)^{n-k}) \quad | \text{Angewandte Logarithmusregel:} \\ \ln(a * b) = \ln(a) + \ln(b)$$

$$\Leftrightarrow \ln(P(X=k)) = \ln\left(\binom{n}{k}\right) + k * \ln(p) + (n - k) * \ln(1 - p) \quad | \text{Angewandte Logarithmusregel:} \\ \ln(a^b) = b * \ln(a)$$

$$\Leftrightarrow P(X = k) = e^{\ln\left(\binom{n}{k}\right) + k * \ln(p) + (n - k) * \ln(1 - p)} \quad | \text{Beide Seiten exponentieren}$$

Vorteile der logarithmischen Darstellung:

Die sehr große Zahl $\binom{n}{k}$ wird nicht direkt mit einer Fließkommazahl verrechnet, sondern durch Anwenden des natürlichen Logarithmus zunächst in eine deutlich kleinere Zahl umgewandelt. Hierdurch werden *OverflowErrors* vermieden. Außerdem ist das Berechnen der Summen von Logarithmen numerisch stabiler als das direkte Berechnen des Produkts. Die Multiplikation einer sehr großen Zahl mit einer sehr kleinen Zahl kann zu erheblichen Ungenauigkeiten führen, die in der logarithmischen Darstellung nicht auftreten.

Die logarithmische Darstellung hat zur Folge, dass die Sonderfälle $p = 0$ und $p = 1$ abgefangen werden müssen, um $\ln(0)$ zu vermeiden (0 ist außerhalb der Definitionsmenge von $\ln()$).

Möglichkeiten zur Modellierung der Regeln

Bei der Simulation eines Laubblasvorgangs muss mit den probabilistischen Regeln angemessen umgegangen werden. Hierfür existieren unter anderem diese Möglichkeiten:

1. Tatsächliche Simulation des Zufalls:

In den folgenden Schritten wird mit den zuvor definierten Bernoulli-Experimenten gearbeitet, die für den Regelfall (Feld 0, A und B sind vorhanden und keine Randfelder) gelten.

- *Schritt 1:* Bernoulli-Experiment 1 wird auf alle x Blätter, die sich auf Feld A befinden, angewendet. Hierfür wird die Zufallsgröße X eingeführt, die angibt, wie viele Blätter von A auf B landen. Anschließlich wird die Binomialverteilung $P(X=k)$ für alle Fälle $0 \leq k \leq x$

mit $n=x$ und $p=P(\text{Bernoulli-Experiment 1})$ berechnet, die hieraus resultierenden Wahrscheinlichkeitswerte werden gespeichert.

- *Schritt 2:* Unter Verwendung einer Zufallsfunktion (wie z.B. `random.choice` in Python) wird einer der Fälle für k zufällig ausgewählt. Die Wahrscheinlichkeiten, zu denen die Zufallsfunktion die einzelnen Fälle auswählt, müssen hierbei den Wahrscheinlichkeiten entsprechen, die über die Binomialverteilung für den jeweiligen Fall ermittelt wurden. Auf die Art wird ermittelt, wie viele Blätter in der Simulation auf Feld B fliegen.
- *Schritt 3:* Auf die $x-k$ Blätter, die nicht auf Feld B fliegen, wird nun analog zu Schritt 1 Bernoulli-Experiment 2 angewendet.
- *Schritt 4:* Analog zu Schritt 2 wird unter Verwendung einer Zufallsfunktion ermittelt, wie viele Blätter in der Simulation auf dem Feld über B landen. Die Blätter, die nicht auf dem Feld über B landen, landen folglich auf dem Feld unter B (einziges übrig bleibendes Szenario).
- *Schritt 5:* Auf die y Blätter, die sich auf Feld B befinden, wird analog zu Schritt 1 Bernoulli-Experiment 3 angewendet.
- *Schritt 6:* Analog zu Schritt 2 wird unter Verwendung einer Zufallsfunktion ermittelt, wie viele Blätter von Feld B auf Feld B liegen bleiben. Die Blätter, die nicht auf Feld B bleiben, müssen folglich auf das Feld rechts von Feld B abgetrieben werden (einziges übrig bleibendes Szenario).

Eine solche Simulation ist zwar am realistischsten, liefert aber bei mehrfachem Ausführen derselben Blasoperation auf identischen Höfen unterschiedliche Ergebnisse und ist daher nicht dafür geeignet, verschiedene Strategien in Hinblick auf ihre Qualität zu vergleichen. Wenn eine Strategie mit dieser Simulation arbeitet, dann muss sie dynamisch auf den aus den vorherigen Blasvorgängen resultierenden Zustand reagieren können, da nicht eindeutig vorhersagbar ist, wie viele Blätter auf welches Feld fliegen werden.

2. **Immer den wahrscheinlichsten Fall wählen:** Um deterministische Ergebnisse zu erhalten, kann man das zuvor vorgestellte Vorgehen so abändern, dass immer der wahrscheinlichste Fall ausgewählt wird. Hiermit gehen jedoch einige Probleme bzw. offene Fragen einher:

- Es ist möglich, dass zwei Fälle gleich wahrscheinlich sind. Welcher Fall sollte dann gewählt werden?
- Die Simulation ist nicht mehr realitätsbezogen, da in der Realität natürlich nicht immer der wahrscheinlichste Fall eintritt.

3. Es ist auch eine Simulation denkbar, die die Wahrscheinlichkeiten nicht auf die einzelnen Blätter, sondern auf „Blattmengen“ anwendet:

Definitionen:

$A_value :=$ Blattanzahl auf A vor Durchführung der Blasoperation

$B_value :=$ Blattanzahl auf B vor Durchführung der Blasoperation

Neue Blattmengen auf den Feldern, die aus dem Blasvorgang resultieren:

$\text{new_B_value} = B_value * (1 - p_{B \rightarrow \text{beyond_B}}) + A_value * p_{A \rightarrow B}$

Seitenabtrieb auf Feld über B = $A_value * p_{A \rightarrow \text{above_B}}$

Seitenabtrieb auf Feld unter B = $A_value * p_{A \rightarrow \text{below_B}}$

Seitenabtrieb auf Feld rechts von B = $B_value * p_{B \rightarrow \text{beyond_B}}$

$\text{new_A_value} = 0$

Da der Erwartungswert in einer binomialverteilten Größe als $n * p$ definiert ist, lässt sich sagen, dass hier die **Erwartungswerte verwendet** werden. Es wird also simuliert, wie die Blätter durchschnittlich flügen, wenn man die simulierte Laubblasoperation sehr oft wiederholen würde. *Vorteile dieser Herangehensweise sind:*

- Da simuliert wird, wie die Blätter durchschnittlich fliegen, ist diese Simulationsart besonders gut für das Entwerfen und Testen einer Strategie geeignet.
- Wird dieselbe Blasoperation mehrfach auf identischen Höfen ausgeführt, dann ist das Ergebnis immer gleich, was das Vergleichen verschiedener Strategien in Hinblick auf ihre Qualität erleichtert.

Nachteile:

- Die Blattanzahl pro Feld bleibt nicht ganzzahlig, sondern wird rational.
- Die Aufgabe legt über die Formulierung „*und ungefähr 10% ist so zu verstehen, dass jedes Blatt mit Wahrscheinlichkeit 10% „beschließt“, das entsprechende Feld aufzusuchen*“ nahe, dass die Wahrscheinlichkeiten auf jedes Blatt einzeln anzuwenden sind und nicht auf „Blatt-Mengen“ – eine Simulation dieser Art ist also eher nicht erwünscht.

Zusammenfassung: Zum Entwickeln und Testen einer Strategie eignet sich diese Simulationsart, sie ist aber keine realistische Simulation der tatsächlichen Anwendung.

Berechnen des Resultats eines Blasvorgangs (Pseudocode)

Die in den Vorkapiteln vorgestellten Verfahren zum Berechnen der aus einem Blasvorgang resultierenden Blattverteilung werden hier als Pseudocode formalisiert. Zunächst ein Algorithmus zur Berechnung der Blasrichtung, die zu einer gegebenen Richtung direction (als Einheitsvektor vorliegend) orthogonal ist:

Algorithmus 1: get_orthogonal_direction (direction)

1. **return** (0,1) wenn direction[1] == 0, ansonsten (1,0)

Algorithmus 2: blase(feld0, blow_direction), entweder über Erwartungswerte oder über die Berechnung der Binomialverteilung und Simulation des Zufalls

1. orthogonal_direction = get_orthogonal_direction(direction)
2. feldA ermitteln, indem der Vektor blow_direction zur Position feld0 addiert wird
3. feldB ermitteln, indem der Vektor blow_direction zur Position feldA addiert wird
4. überprüfen, ob Feld B existiert. Wenn ja:
5. neue Blattanzahl auf Feld B mit entsprechendem Verfahren (Erwartungswerte oder Binomialverteilung-Zufallssimulation) bestimmen
6. Felder über und unter Feld B ermitteln, indem orthogonal_direction zur Position feldB einmal addiert und einmal subtrahiert wird
7. Seitenabtriebsgröße, der auf je eines dieser Felder gerät, mit entspr. Verfahren bestimmen
8. Die Blattanzahlen auf den beiden Feldern über und unter B aktualisieren. Wenn eines der beiden Felder nicht existiert, dann wird das Laub stattdessen zur Anzahl an sich auf B befindenden Blättern addiert

9. Feld rechts von Feld B bzw. gegenüber von Feld A ermitteln, indem blow_direction zur Position feldB addiert wird
10. Seitenabtriebsgröße, der auf dieses Feld gerät, bestimmen
11. Die Blattanzahl auf dem Feld rechts von Feld B aktualisieren. Wenn dieses Feld nicht existiert, dann wird basierend auf dem Parameter s1 entschieden, wie viele Blätter auf Feld B und wie viele auf die Felder über und unter B geweht werden.
12. Ansonsten:
13. Seitenabtriebsgröße, der auf die Felder über und unter Feld A gerät, bestimmen.
14. Blattzahlen auf Feld A und den Feldern über und unter A bestimmen. Wenn eines der Felder über oder unter A nicht existiert, dann verbleibt der entsprechende Seitenabtrieb auf Feld A.

In den folgenden Kapiteln werden verschiedene Lösungsstrategien zum Sammeln des Laubs auf einem Feld Q vorgestellt, die in ihrer Konzeption grundlegende Unterschiede aufweisen. Es werden die Vor- und Nachteile der einzelnen Ansätze aufgezeigt, außerdem wird bewertet, welcher Ansatz am besten die Aufgabenstellung erfüllt.

Trivialer Ansatz: Heuristisches Vorgehen (1. Ansatz)

Hinweis: Es handelt sich bei diesem Ansatz nicht um meinen besten Ansatz. Im nächsten Kapitel auf Ebene 2 stelle ich einen besseren Ansatz vor.

Dieser Ansatz ist konzeptionell am einfachsten: Es wird ein Greedy-Algorithmus durchgeführt, der während des Blasprozesses vor jedem Blasvorgang⁴ ...

1. zunächst alle möglichen Blasoperationen bestimmt,
2. für jede dieser Blasoperationen den aus der Operation hervorgehenden Hof bestimmt,
3. basierend auf einer Heuristik jeden dieser Höfe „bewertet“ bzw. jedem Hof einen Score zuweist und
4. den Blasvorgang auswählt, aus dem der Hof mit dem höchsten Score resultiert.
5. Der ausgewählte Blasvorgang wird anschließend am „tatsächlichen“ Hof durchgeführt.

Das Ziel von Schritt 2 besteht darin, verschiedene Szenarien durchzurechnen, um schließlich die Blasoperation auszuwählen, die voraussichtlich den besten Gesamteffekt auf den Hof hat. Daher werden in Schritt 2 beim Simulieren der Blasvorgänge die Erwartungswerte verwendet, um ein deterministisches Ergebnis zu erhalten, das eine möglichst gute Aussage über die zu erwartende Wirkung des Blasvorgangs trifft.

Man kann sich das so vorstellen, dass der Hausmeister in Schritt 2 verschiedene Szenarien durchrechnet, um sich anschließend für die Blasoperation entscheiden zu können, die am wahrscheinlichsten den besten Gesamteffekt auf den Hof hat.

Im fünften Schritt wird dann der tatsächliche Blasvorgang auf den realen Hof angewendet. Dabei wird der tatsächliche Zufall über die Binomialverteilung simuliert. In diesem Schritt geht es nicht

⁴ Als „Blasprozess“ bezeichne ich den ganzen Prozess, während dem der Hof aufgeräumt wird. Ein Blasprozess besteht aus vielen Blasoperationen / Blasvorgängen.

mehr darum, Vorhersagen zu treffen, sondern darum, die Blasoperation tatsächlich zu simulieren. Hier greift der Hausmeister also tatsächlich zu seinem Laubbläser und führt die Operation durch.

Dieser Ansatz wird nur zum Leeren von Nicht-Randfeldern angewendet, für Randfelder eignet er sich nicht. Im folgenden Teil wird es daher zunächst um das Leeren von Nicht-Randfeldern gehen, sämtliche Blasoperationen, bei denen Feld 0 ein Randfeld ist oder die auf den Rand blasen, werden daher zunächst ausgeklammert. Später wird dann das Leeren des Rands thematisiert.

Kenngrößen zur Bewertung des Zustands eines Hofs

Für die Heuristik in Schritt 3 (siehe Vorkapitel) werden Kenngrößen ermittelt, die eine Aussage darüber treffen, wie „gut“ im Sinne von „erstrebenswert“ ein Hof-Zustand ist.

Die Felder des Hofs sind rasterförmig angeordnet, daher ist eine Verwendung der euklidischen Distanz nicht sinnvoll. Stattdessen wird immer die Manhattan-Distanz verwendet, um den Abstand zwischen zwei Feldern zu bestimmen. Die Manhattan-Distanz zwischen den Punkten P und Q ist definiert als $|Q[0] - P[0]| + |Q[1] - P[1]|$.

1. Kenngröße: Durchschnittliche Blattdistanz zu Feld Q

Angenommen, auf einem Feld P ungleich Q befindet sich ein Blatt B, das auf Feld Q gebracht werden soll. Im Zielzustand befindet sich B auf Q und hat somit den Abstand 0 zu Q. Im Anfangszustand ist der Abstand zwischen B und Q folglich größer als 0 (da P ungleich Q). Eine Sequenz an Blasoperationen, die B von Feld P auf Q befördert, muss den Abstand zwischen B und Q also insgesamt reduzieren. Die Distanz zwischen Blatt B und Feld Q ist also eine Kenngröße, die gut für die Verwendung in einer Greedy-Heuristik geeignet ist, die in jedem Schritt die Blasoperation auswählt, die die Distanz am stärksten reduziert.

Ziel ist es nun natürlich nicht, ein einziges Blatt auf Q zu blasen, sondern alle Blätter des Hofs auf Q zu bringen. Als Kenngröße wird also die durchschnittliche Blattdistanz zwischen allen Blättern auf dem Hof und Q verwendet, die sich mit folgendem Rechenausdruck berechnen lässt:

$$\frac{\sum_x^n \sum_y^n \text{blattverteilung}((x,y)) * \text{manhattan_distance}((x,y), Q)}{\text{sum(blattverteilung)}}$$

$\text{blattverteilung}((x,y))$ repräsentiert dabei die Anzahl an Blättern auf dem Feld am Index (x,y), n ist die Höhe bzw. Breite des quadratischen Hofs und $\text{sum(blattverteilung)}$ repräsentiert die Gesamtanzahl an Blättern auf dem Hof. Der obige Ausdruck berechnet die durchschnittliche Blattdistanz, indem für jedes Feld der Manhattan-Abstand berechnet und mit der Anzahl an Feldern auf dem Feld multipliziert wird. Die Ergebnisse werden über alle Felder gemittelt.

Im angestrebten Endzustand befinden sich alle Blätter auf Feld Q. In diesem Zustand ist die durchschnittliche Blattdistanz 0, faktisch ist er mit dieser Heuristik jedoch nicht erreichbar (mehr dazu später). Daher müssen *Abbruchkriterien* festgelegt werden. Wenn mind. eines der folgenden Abbruchkriterien erfüllt ist, dann wird der Greedy-Algorithmus auf jeden Fall abgebrochen:

- Es gibt keine Blasoperationen mehr, die eine Reduzierung der durchschnittlichen Blattdistanz zur Folge hätten.

- Der Parameter $satisfied_constraint \in [0;1]$ wird eingeführt. Sobald der Anteil des Laubs auf Q am Gesamblaub, das sich auf dem Hof befindet, größer als $satisfied_constraint$ ist, wird der Greedy-Algorithmus abgebrochen.
- Der Parameter $max_operations \in \mathbb{N}$ wird eingeführt. Sobald mindestens $max_operations$ Blasoperationen durchgeführt wurden, wird der Greedy-Algorithmus abgebrochen.

Bei einer Anwendung des Greedy-Algorithmus mit dieser Heuristik-Kenngröße wird das Laub zwar auf Feld Q geblasen, dies geschieht aber sehr unorganisiert:

Angenommen, man hat einen Hof der Größe (9,9) und als Feld Q das Feld in der Mitte bzw. Feld (4,4). Der Greedy-Algorithmus wird als erstes versuchen, sich auf ein von Feld Q relativ weit entferntes Feld zu stellen und Laub Richtung Q zu blasen (siehe Abb. 4, Mitte). Der größte daraus resultierende Laubhaufen in den folgenden Schritten auf direktem Weg Richtung Q geblasen. Durch das direkte Blasen des Laubhaufens Richtung Q kann der Hausmeister mit nur 5 Blasoperationen die Laubmenge auf Feld Q um fast 300 Blätter (Simulation der Blasvorgänge über die Erwartungswerte) erhöhen.

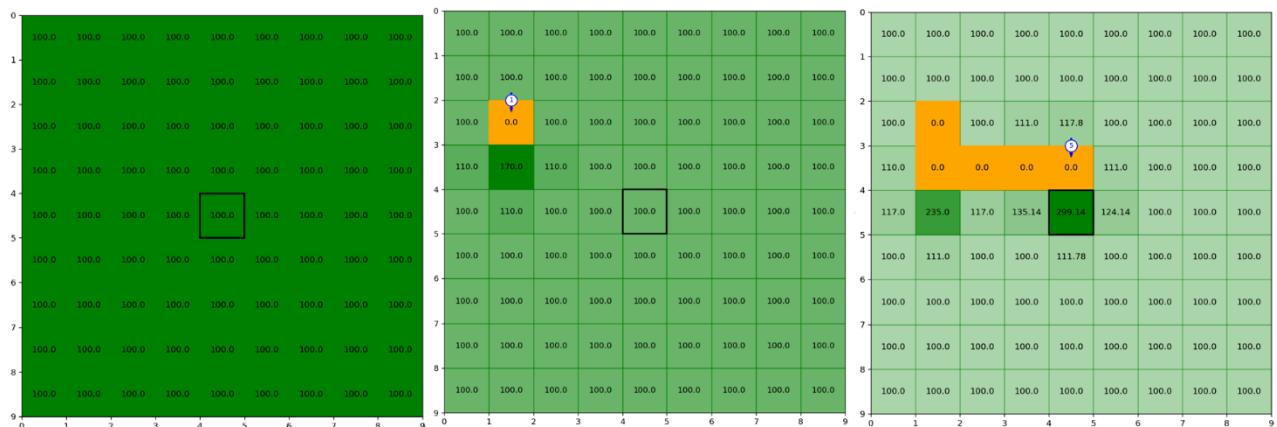


Abbildung 4: Blattverteilung des Hofs bei Anwendung des Greedy-Algorithmus mit durchschnittl. Blattdistanz als Heuristik unter Verwendung der Erwartungswerte nach 0, 1 und 5 Blasoperationen. Feld Q ist schwarz umkastet.

Diese Vorgehensweise ist der verwendeten Heuristik inhärent, da sie immer versucht, am meisten Laub Richtung Q zu blasen, und durch das Blasen des größten Laubhaufens nun mal am meisten Laub transportiert wird.

Nach drei weiteren Blasoperationen wird allerdings sichtbar, was daran problematisch ist: Um Laub von anderen Feldern ebenfalls auf Q zu bringen, muss Laub über bereits geblase Felder geblasen werden (siehe Abb. 5).

Aufgabe 1: Laubmaschen

Teilnahme-ID: 69408

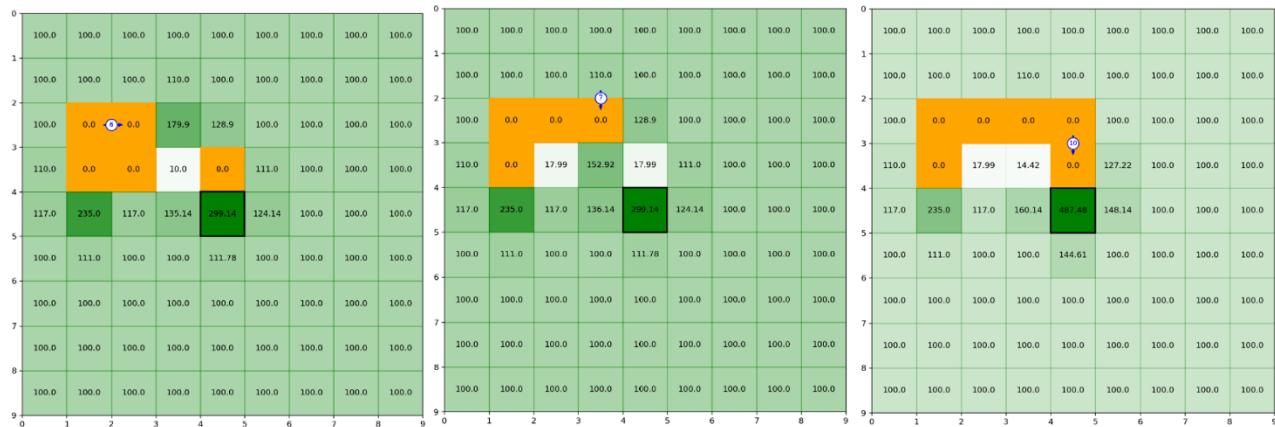
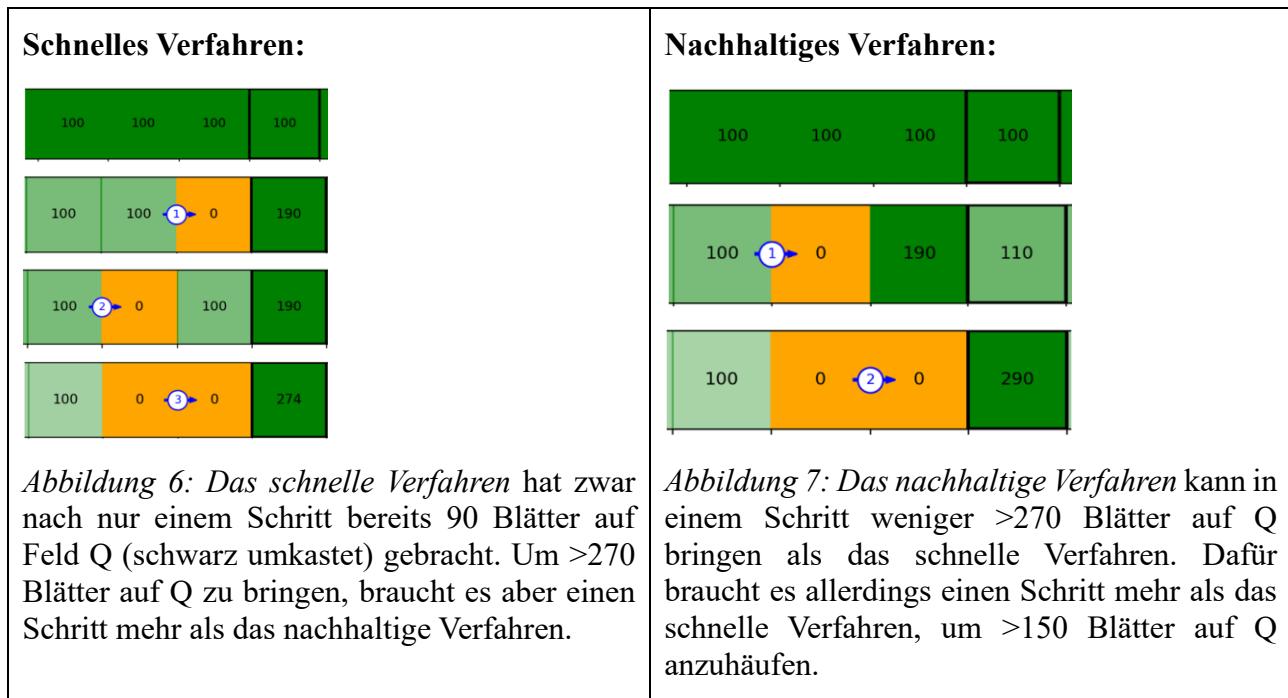


Abbildung 5: Blattverteilung nach 6, 7 und 10 Blasoperationen

Das ist äußerst ungünstig, da die dabei entstehenden Seitenabtriebe dafür sorgen, dass sich auf eigentlich bereits vollständig geleerten Feldern wieder große Mengen an Blättern ansammeln, wie in Abb. 5 sichtbar. Dies sorgt dafür, dass der Hausmeister das Laub dieser Felder erneut auf Q befördern muss – wenn hierbei erneut die durchschnittl. Blattdistanz als Heuristik verwendet wird, entstehen dabei aber wieder Seitenabtriebe auf bereits vollständig geleerte Felder. Wenn nur kleine Mengen an Laub auf Feld Q geblasen werden sollen, können diese Seitenabtriebe ignoriert werden. Soll aber möglichst viel Laub auf Q gelangen, müssen die durch Seitenabtriebe auf bereits geleerte Felder gelangenen Blätter ebenso auf Feld Q geblasen werden müssen, wobei wiederum neue Seitenabtriebe entstehen usw. Dies verlangsamt den Blasprozess deutlich.

Folglich kann der Hausmeister zwar sehr schnell kleinere Mengen an Laub ($< 50\%$ der Gesamtbläumenge) auf Feld Q bringen. Um größere Mengen an Laub ($> 50\%$) auf Feld Q zu konzentrieren, braucht er aber sehr lange. Generell lässt sich beim Blasen von Blättern Richtung Q zwischen einem „schnellen“ und einem „nachhaltigen“ Verfahren unterscheiden, was ich an einem eindimensionalen Beispiel verdeutlichen werden:



Ein schnelles Verfahren zeichnet sich dadurch aus, dass innerhalb von möglichst wenig Schritten das Laub von den Nachbarfeldern von Q auf Q geblasen wird. Ein nachhaltiges Verfahren hingegen leert immer das Feld zuerst, das am weitesten von Q entfernt ist.

Würde man den auf der Blattdistanz-Heuristik basierenden Greedy-Algorithmus auf einen eindimensionalen Hof (einen Hof mit einer Höhe von 1) anwenden, auf dem Q ein Eckfeld ist, dann würde er sich wie ein nachhaltiges Verfahren verhalten. Wie zuvor gezeigt sorgt die Blattdistanz-Heuristik dafür, dass im ersten Schritt auf einem von Q weit entfernten Feld ein Laubhaufen gebildet wird, der anschließend auf direktem Weg Richtung Q geblasen wird (dies wird in Abb. 4 gut sichtbar). Dies entspricht genau der nachhaltigen Vorgehensweise, die in Abb. 7 dargestellt ist.

Die zu betrachtenden Höfe sind allerdings zweidimensional. Auf zweidimensionaler Ebene ist der verwendete Greedy-Algorithmus kein nachhaltiges Verfahren. Ein nachhaltiges Verfahren müsste einen zweidimensionalen Hof leeren, indem es zunächst alle Felder leert, die die Entfernung d von Q haben, anschließend alle Felder leert, die die Entfernung $d-1$ von Q haben usw. mit $d :=$ maximaler Manhattan-Abstand, den ein Feld des Hofs zu Q hat (Randfelder und Felder, die nur über Blasen von einem Randfeld aus geleert werden können, sind hier – wie zuvor erwähnt – nicht berücksichtigt).

Der beschriebene Greedy-Algorithmus geht anders vor: Er leert zunächst ein Feld, das die Entfernung d von Q hat, anschließend leert er ein weiteres Feld, das die Entfernung $d-1$ von Q hat usw. Dies reicht aus, um auf einem eindimensionalen Hof nachhaltig zu leeren, da es hier nur ein Feld gibt, das den Abstand d zu Q hat (wenn Q ein Eckfeld ist). Auf einem zweidimensionalen Hof ist ein nachhaltiges Leeren so aber nicht mehr möglich.

2. Kenngröße: Varianz der Blattdistanzen zu Feld Q

Um die mit der zuvor beschriebenen Greedy-Heuristik einhergehenden Nachteile, brauchen wir eine andere heuristische Kenngröße.

Hof 1:

0	0	0	0	0
0	20	64	40	0
0	20	50	0	0
0	0	0	60	0
0	0	0	0	0

Hof 2:

10	0	0	0	0
0	0	100	0	0
10	0	90	100	0
0	0	0	0	0
0	3	0	77	0

Abbildung 8: Ausschnitte aus zwei Höfen im Vergleich

Auf Hof 1 dürfte die durchschnittliche Blattdistanz einen ähnlichen Wert betragen wie auf Hof 2. Trotzdem ist es auf Hof 1 deutlich einfacher bzw. geht deutlich schneller, > 50% des Laubs auf Feld Q transportieren als auf Hof 2. Dies liegt daran, dass das Laub auf Hof 2 „verstreut“ bzw. weniger geordnet als auf Hof 1 ist. Während es auf Hof 1 einen „Laubcluster“ gibt, dessen Mittelpunkt Feld

Q ist, gibt es auf Hof 2 mehrere separate Laubcluster, zwischen denen leere Felder liegen, was (wie im Vorkapitel erläutert) für den weiteren Blasprozess ungünstig ist.

Der Zustand von Hof 1 ist folglich deutlich erstrebenswerter als der von Hof 2. Um eine Heuristik zu entwickeln, die dies berücksichtigt, müssen wir die „Unordnung“ / „Streuung“ der Blätter auf dem Hof quantifizieren. Als Maß hierfür bietet sich die Standardabweichung der Blattdistanzen zu Q an.

Die Standardabweichung der Elemente einer Menge m der Länge n lässt sich mit folgender Formel

berechnen: $\sqrt{\frac{1}{n} \sum_0^{n-1} (m_m - mean(m))^2}$ Da es bei einem Greedy-Algorithmus aber nur um das

Vergleichen der basierend auf der Kenngröße berechneten „Scores“ geht, kann auch genausogut die kumulierte Varianz statt der Standardabweichung verwendet werden, um Rechenleistung zu sparen (keine Wurzel): $\sum_0^{n-1} (m_m - mean(m))^2$ Die Transformation der Standardabweichung zur kumulierten Varianz ist monotonieerhaltend und beeinflusst das Ergebnis daher nicht. Im folgenden wird diese Heuristik-Kenngröße als „Blattvarianz“ bezeichnet.

Analogie zur Entropie: In der Thermodynamik existiert ein Maß namens Entropie, das die molekulare Unordnung in einem System bzw. die Nutzbarkeit der inneren Energie eines Systems quantifiziert. Auch wenn die zuvor vorgestellte Heuristik nichts mit Thermodynamik zu tun hat, stellt sie ebenfalls ein Maß für die Unordnung der Blätter auf dem Hof dar, das gleichzeitig ein Maß für die „Nutzbarkeit“ der Blattanordnung ist.

Algorithmus 3: Berechnung der kumulierten Blattdistanz

1. blattdistanzen = Leeres Array
2. **für jedes x im Bereich [0 ; n-1]:**
3. **für jedes y im Bereich [0 ; n-1]:**
4. Anzahl_Blätter_auf_Feld = round(blattverteilung((x, y)))
5. **füge Anzahl_Blätter_auf_Feld * manhattan_distance((x, y)) zu blattdistanzen hinzu**
6. **return** varianz(blattdistanzen)

Mit diesem Algorithmus lässt sich grundsätzlich auch die durchschnittliche Blattdistanz annähern, wenn statt der Varianz der Mittelwert bestimmt werden.

Anwendung der Blattvarianz-Heuristik:

Eine reine Anwendung der Blattvarianz-Heuristik, bei der immer die Blasoperation ausgeführt wird, die zur kleinsten Blattvarianz führt, sorgt zunächst nicht dafür, dass Laub auf Q geblasen wird. Stattdessen bewirkt sie, dass die Blattdistanzen der Blätter sich alle demselben Wert annähern (wenn alle Blattdistanzen nahe aneinander sind, dann ist die Abweichung zum Mittelwert folglich am geringsten). Die Blätter würden sich also „ringförmig“ auf Feldern um Q herum anordnen, die alle ähnliche Manhattan-Abstände zu Q haben, was aber natürlich nicht das Ziel des Hausmeisters ist.

Daher muss die Blattvarianz-Heuristik immer der Blattdistanz-Heuristik untergeordnet sein. **Das bedeutet konkret, dass nur die Blasoperationen als mögliche nächste Blasoperation infrage kommen dürfen, die zu einer Verringerung der durchschnittlichen Blattdistanz führen.** Hierdurch wird sichergestellt, dass es sich bei der als nächstes auszuführenden Blasoperation um keine kontraproduktive Operation handelt bzw. das Ziel des Sammelns von Laub auf Q verfolgt wird.

Aus diesen Blasoperationen kann dann die ausgewählt werden, die zur stärksten Verringerung der kumulierten Blattvarianz führt. Hierdurch wird die Unordnung der Blätter so niedrig wie möglich gehalten. Der Nebeneffekt ist hiervon ist aber, dass die Blattvarianz-Heuristik sehr, sehr lange brauchen kann, um nennenswerte Laubmengen auf Q zu versammeln. Wirklich gute Ergebnisse erhält man daher erst, wenn man die Blattvarianz- und Blattdistanz-Heuristik kombiniert.

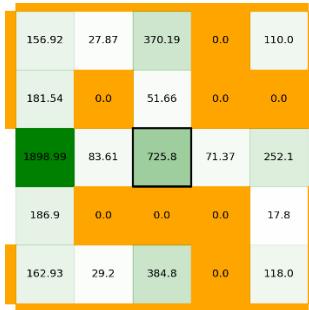


Abbildung 9: Hofausschnitt eines Hofs, auf den die Blattvarianz-Heuristik angewendet wurde

Kombination der Heuristiken

Es werden die Parameter *weight_avg* und *weight_varianz* eingeführt, über die gesteuert werden kann, wie stark die Blattdistanzheuristik (*weight_avg*) und Blattvarianzheuristik (*weight_varianz*) an der Entscheidung, welche Blasoperation als nächstes durchzuführen ist, beteiligt sein sollen.

Da die Blattdistanzheuristik eher einem schnellen Verfahren ähnelt, sorgt ein hoher *weight_avg* und ein niedriger *weight_varianz* Wert dafür, dass Blasoperationen priorisiert werden, die innerhalb von kurzer Zeit viel Laub auf Feld Q bringen, danach aber deutlich weniger Laub auf Feld Q gelangt.

Die Blattvarianzheuristik hingegen ähnelt eher einem nachhaltigen Verfahren, daher sorgt ein niedriger *weight_avg* und ein hoher *weight_varianz* Wert dafür, dass Blasoperationen priorisiert werden, die zunächst eher wenig Laub auf Q bringen, später aber dafür deutlich mehr Laub auf Feld Q gelangt. Dank der Parametrisierung kann der Anwender über *weight_avg* und *weight_varianz* basierend auf seiner Zielsetzung das Verhalten des Algorithmus beeinflussen.

Algorithmus 4: Greedy-Algorithmus mit Kombi-Heuristik

zum Behandeln von Nicht-Randfeldern,

gibt die als nächstes auszuführende Blasoperation zurück

-
1. *current_mw_bd* = Durchschnittliche Blattdistanz am aktuellen Hof ermitteln
 2. *best_score* = -inf
 3. **für jedes x im Bereich [0, n-1]:**
 4. **für jedes x im Bereich [0, n-1]:**
 5. **für jede mögliche Blasrichtung *blow_direction*:**
 6. **falls (x,y) ein Randfeld und (x+blow_direction[0], y+blow_direction[1] ein Randfeld:**
 7. **nächste Schleifeniteration**
 8. Blasoperation *blase((x,y), blow_direction)* an einer Kopie des aktuellen Hofs simulieren
 9. *new_mw_bd* = die aus der Blasoperation hervorgehende durchschnittl. Blattdistanz
 10. **wenn new_mw_bd < current_mw_bd:**
 11. *new_bv* = die aus der Blasoperation hervorgehende durchschnittl. Blattdistanz
 12. *score* = *new_mw_bd* * *weight_avg* + *new_bv* * *weight_varianz*

13. **wenn** score > best_score:
14. best_score = score
15. best_op = In dieser Variable die Blasoperation *blase((x,y,), blow_direction)*
speichern
16. **return** best_op

Mögliche Optimierung:

Die Blasoperationen, bei denen Feld B einen größeren Manhattan-Abstand zu Q hat als Feld A, führen sowieso nicht zu einer Verringerung des durchschnittlichen Blattabstands zu Q und brauchen daher gar nicht erst betrachtet zu werden bzw. es kann direkt in die nächste Schleifeniteration gesprungen werden, wenn auf eine solche Blasoperation getroffen wird.

Behandeln des Rands

Da für Randfelder spezielle Blasregeln gelten, kann der Rand mit der zuvor vorgestellten Heuristik nicht vernünftig geleert werden.

Zunächst ist es sinnvoll, das gesamte sich auf dem Rand befindende Laub auf einem Target-Randfeld bzw. Rand-Zielfeld zu versammeln, von dem aus es dann auf ein Nicht-Randfeld befördert werden kann. (Warum dieses Vorgehen sinnvoll ist, wird später erklärt.) Der schnellste Weg, auf dem dies geschehen kann, besteht darin, das Laub am Rand entlangzublasen.

Festlegung des Rand-Zielfelds:

Als Rand-Zielfeld sollte das Randfeld genommen werden, das den geringsten Abstand zu Q hat. Anschließend an das Sammeln des Randlaubs auf dem Rand-Zielfeld wird das Laub nämlich mit dem zuvor erläuterten Greedy-Verfahren auf Q transferiert, was schneller geht, wenn das ehemalige Randlaub bereitsmöglichst nahe an Q dran ist.

Der Algorithmus zum Leeren des Rands benötigt eine Funktion zur Berechnung der Randdistanz. Hierbei handelt es sich um der Länge der minimalen Verbindungsstrecke zwischen zwei Randfeldern X und Y, die nur über Rand- und Eckfelder läuft. (Es handelt sich also um die Länge des kürzesten Wegs, den man am Rand entlang ablaufen müsste, um von Feld X auf Feld Y zu gelangen):

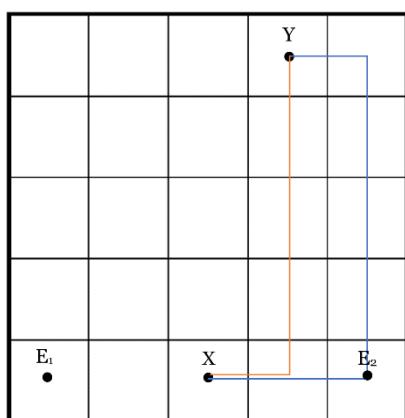


Abbildung 10: Veranschaulichung der Randdistanz: Die Länge der blauen Linie entspricht der Randdistanz. Als Kontrast dazu ist die Linie, deren Länge der Manhattan-Distanz entspricht, orange eingezeichnet. Wie man sieht, entspricht die Manhattan-Distanz nicht der Randdistanz.

Mit einem „Trick“ lässt sich die Randdistanz allerdings über zwei Manhattan-Distanzen berechnen: Wenn sich X und Y *nicht* auf demselben Randsegment befinden, dann kann man sich zunutze machen, dass die Verbindungsgeraden zwischen X und Y, die der Manhattan-Distanz entspricht, auf jeden Fall über eine der Ecken E_1 oder E_2 geht (in Abb. 11 eingezeichnet) – und der Abstand zwischen (E_1 und Y) oder (E_2 und Y) lässt sich wiederum mit der Manhattan-Distanz berechnen. Für diesen Fall gilt also diese Formel:

$$\begin{aligned} \text{randdistanz}(X, Y) = \min & (\text{manhattan_distance}(E_1, X) + \text{manhattan_distance}(E_1, Y), \\ & \text{manhattan_distance}(E_1, X) + \text{manhattan_distance}(E_1, Y)) \end{aligned}$$

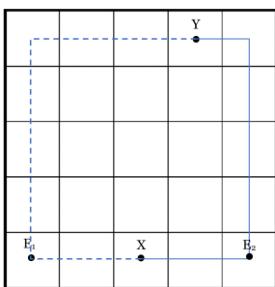


Abbildung 11: Die durchgezogene Linie ist die Linie, deren Länge der tatsächlichen Randdistanz entspricht, die gestrichelte Linie ist die andere infrage kommende Linie

Wenn sich X und Y auf demselben Randsegment befinden, dann ist die Randdistanz zwischen Y und X ganz einfach die Manhattan-Distanz zwischen Y und X.

Um das Randlaub auf dem Rand-Zielfeld zu versammeln wird ein nachhaltiges Verfahren verwendet, das auf einem Greedy-Algorithmus aufbaut. Dieser Algorithmus speichert alle Randfelder, die er bereits geleert hat, in einer Liste. Er wählt in jedem Schritt das noch nicht geleerte Randfeld aus, das am weitesten vom Rand-Zielfeld entfernt ist, und bläst das sich auf dem gewählten Feld befindende Laub den Rand entlang auf ein anderes Randfeld, das näher am Rand-Zielfeld ist:

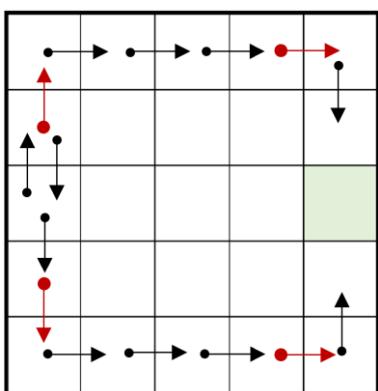


Abbildung 12: Entlangblasen des Laubs am Rand, um es auf einem Rand-Zielfeld zu versammeln

Algorithmus 5: (grobe Skizzierung des) Greedy-Algorithmus zum Sammeln des Laubs auf dem Rand-Zielfeld,

gibt die als nächstes auszuführende Blasoperation zurück

Parameter: `edge_fields_to_clear` = Eine Liste mit den noch zu leerenden Randfeldern (vor dem ersten Blasvorgang enthält diese Liste alle Randfelder)

Parameter: `Rand-Zielfeld` = Das Randfeld, auf dem das Randlaub versammelt werden soll, bevor es schließlich auf ein Nicht-Randfeld geblasen wird.

1. `field_to_clear` = Das Feld in `edge_fields_to_clear` finden, das die größte Randdistanz zum Rand-Zielfeld hat
2. `target_field` = das Nachbarfeld von `field_to_clear`, das ein Randfeld ist, in `edge_fields_to_clear` ist und am nächsten am Rand-Zielfeld dran ist (niedrigste Randdistanz)
3. **return** die Blasoperation, die einen Großteil des Laubs vom Feld `field_to_clear` auf das Feld `target_field` bläst

Um dabei das Laub eines Randfelds auf ein benachbartes Randfeld (=“Operationszielfeld“) zu transferieren, muss nur einmal von dem Randfeld aus, das sich neben dem zu leerenden Randfeld und gegenüber vom Operationszielfeld befindet, auf das zu leerende Randfeld blasen:



Die Eckfelder lassen sich jedoch nicht mit einem Blasen vollständig leeren, da bei dem Blasen auf ein Eckfeld immer nur ein Teil des Laubs auf das Operationszielfeld gelangt. Ein Eckfeld hat nur zwei Nachbarfelder, von denen eines das Operationszielfeld ist (bzw. das Feld, auf das der Hausmeister das Laub befördern will). Das andere Nachbarfeld des Eckfelds ist das Feld, auf das sich der Hausmeister stellen muss, um Laub vom Eckfeld auf das Operationszielfeld zu befördern. Die Blasoperationen, die Eckfelder leeren, sind in Abb. 10 rot eingezeichnet.

Diese Blasoperation muss der Hausmeister dann solange ausführen, bis die Blattanzahl auf dem Eckfeld keine wesentliche Laubmenge mehr liegt bzw. die Restlaubmenge einen bestimmten Toleranzwert unterschreitet. In der Implementierung wird dieser Toleranzwert automatisch auf `startwert * (1-satisfied-constraint)` gesetzt (`startwert` ist die Blattanzahl, die sich zu Beginn auf jedem einzelnen Feld befindet, und `satisfied_constraint` ist die Abbruchbedingung des Programms bzw. der Parameter legt fest, wie hoch der Anteil des Laubs auf Q am Gesamtblaub sein muss, damit das Programm auf jeden Fall abgebrochen wird). Dies ist sinnvoll, da die Anzahl des aus den Ecken zu holenden Laubs natürlich davon abhängt, wie viele Blätter man denn überhaupt auf Q bringen möchte bzw. muss.

All dies wirft die Frage auf, ob es möglich ist, Ecken und die Ränder vollständig zu leeren. Mit dieser Frage werde ich mich in Ansatz 2 beschäftigen.

Transferieren des Laubs vom Rand-Zielfeld auf ein Nicht-Randfeld:

Nach Vollendung der zuvor erläuterten Schritte ist das Laub vollständig auf dem Rand-Zielfeld versammelt und muss nun auf ein Nicht-Randfeld gebracht werden.

Da sich der Hausmeister nur auf Felder des Hofs stellen darf, ist das „direkte“ Blasen von Laub von einem Rand- auf ein Nicht-Randfeld nicht möglich. Stattdessen können Randfelder geleert werden,

imdem auf den Nachbar-Randfeldern des Randfels orthogonal zum Rand das Laub solange hin- und hergeblasen wird, bis die Laubemenge auf den Randfeldern einen bestimmten Betrag unterschreitet.

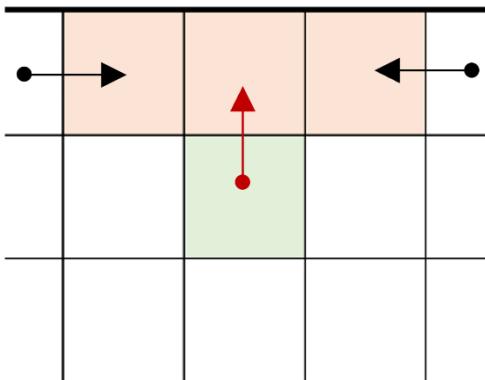


Abbildung 13: Blasoperationen zum Leeren des Rands

Bei jeder der beiden schwarz eingezeichneten Blasoperationen gelangt über den Seitenabtrieb Laub von den rot eingefärbten Feldern auf das grün unterlegte Feld. Um also das Laub von den drei roten Randfeldern auf das grüne Nicht-Randfeld zu befördern, müssen die schwarzen Blasoperationen mehrfach ausgeführt werden. Hierbei akkumuliert zunehmend Laub auf dem rot eingefärbten Feld über dem grün eingefärbten Feld, was den Gesamtprozess verlangsamt. Um dem entgegenzuwirken, kann zwischendurch die rot eingezeichnete Blasoperation eingeschoben werden, die zwar kein Laub auf das grüne Feld befördert, aber stattdessen dafür sorgt, dass bei den folgenden schwarzen Blasoperationen mehr Laub auf das grüne Feld gelangt.

Da bei jeder schwarzen Blasoperation ein bestimmter Prozentsatz des Laubs, das sich auf den roten Feldern, auf das grün eingefärbte Feld geblasen wird, ist das Vorgehen zum Leeren des Randfelds effektiver (d.h. der absolute Wert an pro Blasoperation auf das grüne Feld geblasenen Laubs ist größer), je mehr Laub sich auf den roten Feldern befindet. Hieraus lässt sich schlussfolgern, dass zunächst das gesamte Randlaub auf einem Randfeld versammeln werden sollte, bevor dieses Randfeld dann systematisch geleert wird.

Werden die zuvor erläuterten Vorgänge zu Algorithmus hinzugefügt, dann ergibt sich dieser Gesamt-Algorithmus zum Leeren des Rands. Die zuvor erläuterten Mechanismen sind dort entweder direkt oder indirekt umgesetzt, stellenweise ist der Pseudocode zur Verdeutlichung mit Kommentaren versehen:

Algorithmus 6: Greedy-Algorithmus zum Leeren des Rands, gibt die als nächstes auszuführende Blasoperation zurück

Globale Liste: `edge_fields_to_clear` = Eine Liste mit den noch zu leerenden Randfeldern (vor dem ersten Blasvorgang enthält diese Liste alle Randfelder)

Parameter: `Rand-Zielfeld` = Das Randfeld, auf dem das Randlaub versammelt werden soll, bevor es schließlich auf ein Nicht-Randfeld geblasen wird.

Globale Liste: `edge_target_field_neighbors` = Liste, in der die Nachbarn des Rand-Zielfelds gespeichert werden

Globale Integer-Variable: `clear_edge_last_field_index` = Variable, die speichert, welcher Schritt zur Leerung des Rands als nächstes ausgeführt werden soll

1. **field_to_clear** = Wähle das Feld aus **edge_fields_to_clear** aus, das die größte Randdistanz zum Rand-Zielfeld hat
2. **Falls** sich noch eine wesentliche Laubmenge auf Feld **field_to_clear** befindet:
3. **Wenn** **field_to_clear** ein direkter Nachbar des Rand-Zielfelds ist:
 Füge **field_to_clear** **zu** **edge_target_field_neighbors** **hinzu** (**falls** noch nicht in Liste)
5. **Falls** **field_to_clear** Nachbarfeld des Rand-Zielfelds ist:
 Aktiviere das zyklische Blasverfahren, das das Laub vom Rand-Zielfeld hin und herbläst, um es über die Seitenabtriebe nach und nach auf ein Nicht-Randfeld zu bringen:
 Füge die Nachbarfelder des Rand-Zielfelds zur Liste der zu leerenden Felder hinzu
7. Speichere die in diesem Schritt auszuführende Operation des zyklischen Randleerungsverfahrens.
8. Rotiere durch die Operationen des zyklischen Verfahrens, indem festgelegt wird, dass im nächsten Schritt die nächste Operation des zyklischen Randleerungsverfahrens durchgeführt wird (hierfür wird **clear_edge_last_field_index** um 1 erhöht, sobald es den Wert 3 erreicht wird es auf 0 zurückgesetzt). Die Operationen des zyklischen Randleerungsverfahrens sind die in Abb. 13 eingezeichneten Blasoperationen, das in Abb. 13 rot eingefärbte und sich über dem grünen Feld befindende Feld ist dabei das Rand-Zielfeld.
9. **return** die in Schritt 7 gespeicherte Blasoperation
10. **target_field** = Das Nachbarfeld von **field_to_clear** finden, dass am nächsten am Rand-Zielfeld ist (Rand-Distanz)
11. **Falls** **field_to_clear** **kein** Eckfeld:
 Entferne **field_to_clear** **aus** **edge_fields_to_clear** (Erläuterung: Nicht-Eckfelder können durch einmaliges Blasen geleert werden)
13. **return** die Blasoperation, die am meisten Laub von **field_to_clear** auf **target_field** bringt
14. **Andernfalls:**
15. **Entferne** **field_to_clear** **aus** **edge_fields_to_clear**
16. **Führe Algorithmus 6 erneut aus**

Gesamt-Algorithmus

Der Gesamt-Algorithmus setzt sich aus dem Greedy-Algorithmus zum Leeren des Rands (Algorithmus 6) und dem Greedy-Algorithmus zum Leeren der Nicht-Randfelder (Algorithmus 4) aus:

Algorithmus 7: Gesamt-Algorithmus von Ansatz 1

1. **while** keine Abbruchbedingung erfüllt
2. **solange, bis Algorithmus 6 keine Blasoperationen mehr findet** (dieser Fall tritt ein, wenn sich auf keinem Randfeld mehr eine wesentliche Blattmenge befindet):
 blasoperation = **führe Algorithmus 6 aus und speichere Rückgabe**
 führe **Blasoperation** **blasoperation** **aus**
 falls mind. eine Abbruchbedingung erfüllt:
 return

7. **solange, bis Algorithmus 4 keine Blasoperationen mehr findet** (dieser Fall tritt ein, wenn es keine Blasoperation mehr gibt, die die Laubmenge auf Feld Q erhöht):
8. **blasoperation = führe Algorithmus 4 aus und speichere Rückgabe**
9. **führe Blasoperation blasoperation aus**
10. **falls** mind. eine Abbruchbedingung erfüllt:
11. **return**

Der Gesamt-Algorithmus nimmt folgende Parameter:

- Q (Tupel): Ein Tupel, das den Index von Feld Q angibt
- hof_size (Tupel): Ein Tupel, das die Größe des Hofs angibt
- startwert (ganze Zahl): Gibt die Anzahl an Blättern an, die sich zu Beginn auf den Feldern befindet
- use_binomial (Wahrheitswert): Gibt an, ob bei der Simulation der Blasvorgänge am tatsächlichen Hof in Algorithmus 7 die Erwartungswerte verwendet werden sollen oder ob die tatsächlichen Wahrscheinlichkeiten zu simulieren sind
- satisfied_constraint, max_operations, weight_avg, weight_varianz: In den Vorkapiteln erläutert

Laufzeit

Bei der Bestimmung der asymptotischen Laufzeit werden alle der oben genannten Parameter bis auf hof_size als Konstanten behandelt. Besonders bei satisfied_constraint und max_operations ist dies sinnvoll, da es sich hierbei um obere Schranken handelt, die beliebig gesetzt werden können. Natürlich ist die tatsächliche Evaluierungszeit des Programms niedriger, wenn die Schranken niedrig gesetzt sind. Viel interessanter ist die asymptotische Laufzeit in Abhängigkeit zur Feldlänge bzw. -breite n.

Die Laufzeit in Abhängigkeit von n wird dabei eindeutig von Algorithmus 4 dominiert, der bei jeder Ausführung einmal über alle Felder iteriert und somit eine Laufzeit von $O(n^2)$. Mit Blick auf Algorithmus 7 lässt sich sagen, dass Algorithmus 4 im Worst-Case-Szenario max_operations Male ausgeführt wird. Das Worst-Case-Szenario tritt dabei auf, wenn Algorithmus 7 ausschließlich Algorithmus 4 und nie Algorithmus 5 ausführt und Algorithmus 7 erst beim Erreichen von max_operations durchgeföhrten Operationen abgebrochen wird (es darf keine andere Abbruchbedingung zuvor eintreten). Wenn max_operations doch nicht als Konstante betrachtet wird, dann lässt sich also sagen, dass die Laufzeit des Gesamtverfahrens im Worst-Case-Szenario eine Laufzeit von $O(n^2 * \text{max_operations})$ hat. Diese Laufzeit ist leider nicht besonders gut, es handelt sich hierbei aber nicht um den einzigen Ansatz, den ich entwickelt habe (Ansatz 2 hat eine bessere Laufzeit).

Kritik am Algorithmus

Der Vorteil des ersten Ansatzes besteht darin, dass über die Parameter satisfied_constraint, max_operations, weight_avg und weight_varianz flexibel gesteuert werden kann, wie viele Blätter auf Feld Q gebracht werden sollen, wie viele Operationen maximal ausgeführt werden, und wie schnell die Blätter auf Feld Q gelangen sollen.

Es ist jedoch fraglich, ob der Algorithmus den Anforderungen des Hausmeisters entspricht. In der Aufgabenstellung wünscht sich der Hausmeister *nicht, möglichst schnell Laub auf Q zu bringen*. Stattdessen wünscht er sich explizit, dass die Laubmenge auf Feld Q zu maximieren ist. Der Algorithmus des 1. Ansatz garantiert dabei nicht, dass die Gesamtblaubmenge auf allen Feldern außer von Q gegen Null konvergiert, wenn sehr viele Blasoperationen durchgeführt werden (und zur Modellierung der Blasoperationen durchweg die Erwartungswerte verwendet werden). Die Formulierung „Maximierung der Laubmenge auf Q“ legt aber nahe, dass ein solches Verfahren gesucht ist, das die Laubmenge auf Q maximiert und die kumulierte Laubmenge auf den anderen Feldern minimiert bzw. gegen Null konvergieren lässt. Im nächsten Kapitel werde ich meinen 2. Ansatz vorstellen, bei dem es sich um ein solches Verfahren handelt und der noch dazu eine deutlich bessere Laufzeit als mein 1. Ansatz hat.

Wegen der kaum systematischen Herangehensweise gelingt es außerdem gerade bei kleinen Höfen nicht, große Mengen an Laub auf Feld Q zu versammeln.

Besseres Vorgehen: Generalisierte Ablaufpläne (2. Ansatz)

Ziel

In der Aufgabenstellung heißt es: „Das Ziel des Hausmeisters ist, das (...) Laub vom gesamten Schulhof auf ein einziges Planquadrat Q zu konzentrieren“. Hieraus lässt sich schlussfolgern, dass **das Versammeln von möglichst viel Laub auf Feld Q für den Hausmeister am wichtigsten ist** und alle anderen Aspekte wie z.B. Anzahl an durchzuführenden Blasoperationen (natürlich ebenfalls nicht unwichtig, aber) zweitrangig sind. Ich werde daher in den folgenden Kapiteln ein 2. Verfahren vorstellen, das sich in seiner Konzeption grundlegend vom 1. Ansatz unterscheidet.

Konkret soll für das 2. Verfahren folgendes gelten (wenn zur Modellierung der Wahrscheinlichkeiten die Erwartungswerte verwendet werden):

- In der Lösung sollen so viele Felder wie möglich (außer natürlich Q) vollständig geleert sein. Diese Aussage wird unter der Voraussetzung getroffen, dass zur Modellierung des Zufalls die Erwartungswerte verwendet werden. Das bedeutet: Es ist wirklich gemeint, dass so viele Felder wie möglich vollständig geleert werden. (Auf einem vollständig geleerten Feld ist die Laubmenge im Gegensatz zu einem *asymptotisch geleerten Feld* nicht nur asymptotisch Null bzw. geht gegen Null, sondern ist tatsächlich Null).
- Es können nicht alle Felder vollständig geleert werden (siehe hierzu nächstes Kapitel). Für die Felder, die nicht vollständig geleert werden können, wird ein Parameter tolerated_amount eingeführt, der angibt, wie viele Blätter auf einem solchen Feld maximal „toleriert“ werden.
- Bei dem Algorithmus soll es sich folglich um einen Conquer-Algorithmus handeln, der (anders als das 1. Verfahren) nicht unorganisiert versucht, immer wieder auf Feld Q zu blasen, sondern stattdessen einen kontinuierlich größer werdenden Bereich des Hofes vollständig und endgültig leert. Es sind nämlich nur auf die Art die beiden zuvor genannten Ziele erreichbar.
- Auch wenn die Minimierung der Anzahl benötigter Blasoperationen kein primäres Ziel ist, soll das Verfahren den Hof trotzdem so leeren, dass nicht unnötig viel Aufwand für den Hausmeister

entsteht. Unser Ziel ist es, den Hof zu leeren, ohne dass der Hausmeister den Eindruck hat, er müsse sich in einem Blasorchester wiederfinden!

- Anders als im 1. Ansatz wird in diesem Verfahren nicht mit einer volldynamischen Greedy-Heuristik gearbeitet. Die algorithmische Umsetzung des Verfahrens soll dennoch so erfolgen, dass – wie im 1. Ansatz – eine Funktion existiert, die entscheidet, welcher Blasvorgang als nächstes auszuführen ist (dies passt am besten zu der in der Aufgabe gegebenen Formulierung „Die Strategie entscheidet vor jedem Blasvorgang, auf welches Feld er sich stellen und in welche Richtung er von dort jeweils blasen soll“). Für das im Folgenden vorgestellte Verfahren benötigt man allerdings einen kognitiv deutlich fitteren Hausmeister, da er sich deutlich mehr Informationen „merken muss“.

Auf welchen Höfen kann das Laub nicht auf Q geblasen werden?

Für Höfe mit $n < 2$ existiert offensichtlich keine Lösung, da solche Höfe nur aus Eckfeldern bestehen.

In einem Hof der Maße (3,3) gibt es zwar ein mögliches Feld Q (das Feld in der Mitte bzw. mit den Koordinaten (1,1)). Es ist aber nicht möglich, Laub von einem Randfeld auf das Feld in der Mitte zu bringen, da es keine auf diesen Hof anwendbare Regel gibt, mit der Laub von einem Randfeld auf ein Nicht-Randfeld gebracht werden kann. **Folglich ist es auch nicht möglich, Laub von einem Eckfeld auf das Feld in der Mitte zu bringen**, da Laub aus Ecken nur auf Randfelder gebracht werden kann (ein Eckfeld hat nur Randfelder als Nachbarn).

In einem solchen Hof ist es daher am besten, gar keine Blasoperationen auszuführen, da der beste erreichbare Zustand bereits vorliegt.

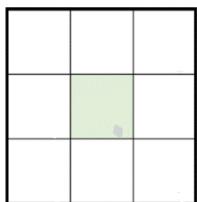
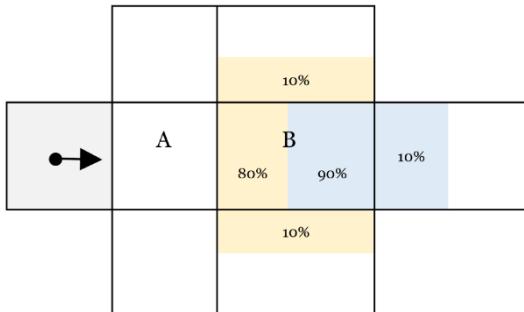


Abbildung 14: Hof mit den Maßen (3,3). Es gibt keine Möglichkeit, Laub auf das Feld #4 in der Mitte zu befördern - egal wie geblasen wird.

Auf allen anderen Höfen ist es möglich, die im Vorkapitel beschriebenen Ziele zu erreichen.

Zugrundeliegende Strategie

Zunächst müssen wir uns von dem Gedanken lösen, dass es ein volldynamisches Verfahren braucht, um das Problem zu lösen. Der Umstand, dass die Blasoperation nichtdeterministisch sind, kann einen zu dieser Idee verleiten, dabei wird allerdings übersehen, dass nur auf manchen Feldern die Änderung der Blattmenge vom Zufall abhängt:



So ist die z.B. die Wahrscheinlichkeit dafür, dass ein Blatt, das sich vor dem Blasvorgang auf Feld A befindet, auf Feld A verbleibt, 0% (siehe Abb. oben). **Feld A wird nach dem Blasvorgang also auf jeden Fall leer sein** (unter der Voraussetzung, dass Feld B im Hof existiert). Es ist außerdem zu 100% sicher, dass sich die Laubmenge auf den Feldern unter und über A nicht durch die Blasoperation verändern wird. Hieraus ergibt sich, dass **große Teile des Hofes vollständig (nicht nur asymptotisch) geleert werden können**, indem das Laub Reihe für Reihe bzw. Spalte für Spalte an eine Hofkante geblasen wird. Das folgende Beispiel zeigt einen Hof der Größe (5,5), der dieses Vorgehen veranschaulicht: Zunächst wird die 2. Spalte vollständig geleert, danach die 3. Spalte usw. bis sich nur noch in der 1. und der 5. Spalte Laub befindet.

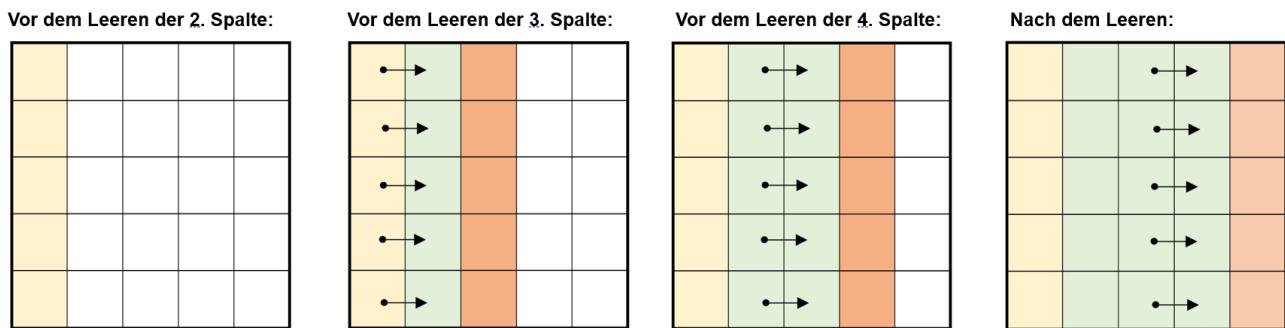
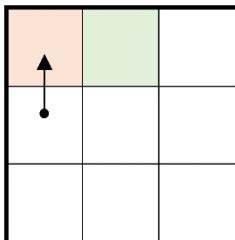


Abbildung 15: Hof der Größe (5,5), auf dem das Verfahren zum vollständigen Leeren aller Felder (bis auf zwei Zeilen / Spalten mit Randfeldern) visualisiert ist. In jedem Hof sind die in der vorherigen Phase durchgeführten Blasprozesse eingezeichnet. Die grünen Felder sind nach den Blasprozessen vollständig geleert. Das von den grünen Feldern heruntergeblasene Laub befindet sich größtenteils auf den orangenen Feldern, teilweise auch auf den weißen Feldern. Die Blattanzahlen auf den gelb eingefärbten Feldern bleiben unverändert.

Dieses Verfahren leert die Felder, die sich zwischen der 1. und 5. Spalte befinden. Er ist ein wesentlicher Bestandteil des im 2. Ansatz verwendeten Conquer-Algorithmus. Die Leerung der grünen Felder erfolgt zu einer Wahrscheinlichkeit von 100%; es wäre daher unsinnig, hier ein dynamisches Verfahren zu verwenden.

Es gibt jedoch andere Fälle, in denen sich ein dynamisches Verfahren nicht vermeiden lässt.

Beispiel hierfür: Leeren eines Eckfelds



Ein Eckfeld lässt sich nur durch das kontinuierliche Blasen in die jeweilige Ecke leeren. Wenn dabei die tatsächlichen Zufälle simuliert werden, dann wandert beim Blasen jedes Blatt, dass sich auf dem Eckfeld befindet, zu einer bestimmten Wahrscheinlichkeit auf das Randfeld neben dem Eckfeld, auf dem nicht der Hausmeister steht. Die Anzahl an Blättern, die das Eckfeld verlassen, ist dabei vom Zufall abhängig, was ein dynamisches Verfahren erfordert.

Definitionen: Generalisierte Ablaufpläne und Muster

Eine Sequenz an Blasoperationen, die den zuvor definierten Zielen gerecht wird, muss also auf jeden Fall aus nicht-dynamischen Phasen, während denen eine feste, „vorprogrammierte“ Abfolge an Blasoperationen durchgeführt wird, und aus dynamischen Phasen, während denen Blasoperationen aus einer begrenzten Operationsmenge basierend auf der tatsächlichen Veränderung der Blattverteilung (die dann, wenn die tatsächlichen Zufälle über die Binomialverteilung simuliert werden, nicht eindeutig vorhersagbar ist) solange ausgeführt werden, bis ein bestimmtes Ziel erreicht ist.

Um diesem Dualismus gerecht zu werden, definiere ich im Folgenden zwei theoretische Konzepte, die in der Umsetzung beide implementiert werden:

Definition „Muster“: Als *Muster* wird im Folgenden eine Operation bezeichnet, das eine aus einer oder mehreren Blasoperationen bestehende Sequenz an Blasoperationen und anderen Mustern speichert. Ein Muster verfolgt das Ziel, Laub von einem oder mehreren *source*-Feldern auf ein *target*-Feld zu schaffen. „Das Muster wird ausgeführt“ bedeutet konkret, dass die vom Muster gespeicherten Blasoperationen ausgeführt werden.

Je öfter das Muster wiederholt wird, desto mehr Laub wird von den *source*-Feldern auf das *target*-Feld geschafft. Bei einer gegen unendlich gehenden Anzahl an Wiederholungen des Musters geht die Blattanzahl auf den *source-Feldern* daher gegen 0, die *source*-Felder werden also asymptotisch geleert.

Es wird ein Parameter *tolerated_amount* $\in \mathbb{Q} > 0$ eingeführt, der angibt, wie hoch die höchstens tolerierte Blattanzahl auf den zu leerenden *source*-Feldern ist. Bei Ausführung werden das Muster so lange wiederholt, bis die maximale Blattanzahl auf einem der *source*-Felder kleiner gleich *tolerated_amount* ist. Ein Muster ist also dynamisch.

Wird *tolerated_amount* auf einen sehr niedrigen Wert gesetzt, dann wird das Muster folglich sehr oft wiederholt werden. Bei einem hohen Wert wird das Muster nicht so oft wiederholt, allerdings werden dann auch weniger Blätter von den *source*-Feldern entfernt.

Die Verwendung eines Musters ist immer dann sinnvoll, wenn ein Feld oder mehrere nebeneinanderliegende nicht alle gleichzeitig in einen vollständig geleerten Zustand gebracht werden können.

Ein Muster muss so definiert werden, dass beim Ausführen des Musters nur auf den source-Feldern und dem Feld *target* die Blattzahlen verändert werden (alles andere würde dem Conquer-Prinzip widersprechen, das dem 2. Ansatz zugrundeliegt).

Definition „Generalisierter Ablaufplan“: Ein generalisierter Ablaufplan speichert eine Sequenz an Blasoperationen und Mustern. Die Verwendung eines generalisierten Ablaufplans ist dann sinnvoll, wenn zum Erreichen eines bestimmten Ziels sowohl Phasen, in denen der nächste Blasvorgang dynamisch auszuwählen ist, als auch nicht-dynamische Phasen, in denen die durchzuführenden Blasoperationen bereits vorher klar sind, durchgeführt werden müssen.

Die in nicht-dynamischen Phasen durchzuführenden Blasoperationen können direkt als solche im generalisierten Ablaufplan gespeichert werden. Um dynamische Phasen zu speichern, können Muster verwendet werden, die dynamisch arbeiten, wenn sie ausgeführt werden.

Ein generalisierter Ablaufplan ist dabei etwas, was nicht während dem Blasprozess, sondern vor dem Durchführen der ersten Blasoperation erstellt wird. Während der eigentlichen Simulation des Blasprozesses werden die im generalisierten Ablaufplan enthaltenen Blasoperationen (nicht-dynamisch) und Muster (dynamisch) dann nur noch ausgeführt. Das im Folgenden vertiefte Lösungsverfahren funktioniert genau auf diese Art und Weise.

Algorithmus 8: Grobe Skizzierung des Algorithmus von Ansatz 2

1. Erstellen des generalisierten Programmablaufplans (der Hausmeister muss sich diesen gut einprägen)
2. Durchführen der Blasoperationen und Muster aus dem in Schritt 1 erstellten generalisierten Programmablaufplan

Können die Rand- und Eckfelder vollständig geleert werden?

Voraussetzung: Zur Modellierung der Wahrscheinlichkeiten werden die Erwartungswerte verwendet (nur unter dieser Voraussetzung macht die Frage Sinn).

Eckfelder können nicht vollständig geleert werden, sondern nur asymptotisch geleert werden: Beim Blasen in eine Ecke verlässt immer nur ein bestimmter Anteil kleiner als 100% der sich auf dem Eckfeld befindenden Blätter das Eckfeld.

Fast alle Randfelder können vollständig geleert werden, es bleibt aber mind. 1 Randfeld übrig, das nicht vollständig geleert werden kann: Es ist möglich, dass Laub (wie bei der Erläuterung des 1. Ansatz zu beschreiben) am Rand entlangzublasen. Hierbei werden alle Randfelder bis auf zwei vollständig geleert (Eckfelder nicht miteinbezogen): Das Randfeld, auf dem das Laub akkumuliert wird, und eines seiner Nachbarfelder können nur asymptotische geleert werden. (Wenn das

Randfeld, auf dem das Laub akkumuliert wird, Nachbar eines Eckfelds ist, dann ist es auch möglich, alle Randfelder bis auf eines zu leeren (Eckfelder nicht miteinbezogen)).

Muster zum Leeren von Rand- und Eckfeldern

Zum Leeren der Ecke links oben wird dieses Muster definiert, das Laub vom Source-Feld (0,0) auf das Target-Feld (1,0) schafft:

- blase(Feld (0,1), (1,0))

In der Darstellung rechts ist das source-Feld rot markiert und das target-Feld grün markiert.

(0,0)	(1,0)	(2,0)
(0,1)	(1,1)	(2,1)
(0,0)	(1,2)	(2,2)

Das Muster lässt sich auf jede Ecke anwenden, wenn es entsprechend gespiegelt und gedreht wird. Hierfür wird ein Algorithmus entwickelt:

Algorithmus 9: Algorithmus, der ein Muster zum Leeren einer Ecke ermittelt

Parameter: source_corner_field = Der Index des Eckfelds, das geleert werden soll

Parameter: target_field = Der Index des Randfelds, auf das das Laub geblasen werden soll

- orthogonal_direction = Ermitteln der Richtung orthogonal zum Vektor (target_field-source_field) mithilfe von Algorithmus 1
- feld0 = Finde das Feld, von dem aus in die Ecke geblasen werden muss, um Laub aus der Ecke zu befördern. Für feld0 kommen die Felder target_field-orthogonal_direction und target_field+orthogonal_direction infrage, die durchprobiert werden.
- return** ein Muster mit folgenden Eigenschaften:
 - Das Source-Feld des Musters ist source_corner_field
 - Das Target-Feld ist target_field
 - Das Muster speichert eine Blasoperation, die von Feld 0 aus in die zu leerende Ecke bläst

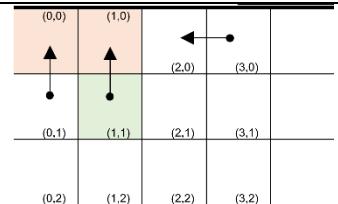
Zum Leeren des Randfelds wird dieses Muster definiert, das Laub von den Source-Feldern (1,0), (2,0) und (3,0) auf das Target-Feld (0,1) schafft:

- blase((2,1), (0,-1))
- blase((0,0),(1,0))
- blase((4,0),(-1,0))

(0,0)	(1,0)	(2,0)	(3,0)
(0,1)	(1,1)	(2,1)	(3,1)
(0,2)	(1,2)	(2,2)	(3,2)

Wenn das zu leerende Randfeld ein Nachbarfeld eines Eckfelds ist, muss das Muster leicht abgeändert. Es wird ein neues Muster definiert, das Laub von den Source-Feldern (0,0), (1,0) und (2,0) auf das Target-Feld (0,1) schafft:

1. blase((1,1), (0,-1))
2. blase((0,1),(0,-1))
3. blase((3,0),(-1,0))



Auch dieses Muster lassen sich wieder auf jedes Randfeld anwenden, wenn es entsprechend gespiegelt und gedreht wird. Hierfür wird ein Algorithmus entwickelt, der aber nicht mehr in Pseudocode formalisiert wird. Das Prinzip, basierend auf dem Randfelder gelehrt werden, sollte inzwischen klar geworden sein.

Bauen des generalisierten Ablaufplans

Dieses Kapitel beschäftigt sich mit den Algorithmen, die den generalisierten Ablaufplan zusammenbauen (Schritt 1 von Algorithmus 8). Grundsätzlich lässt sich sagen, dass der Ablaufplan immer aus diesen **4 Phasen** besteht:

1. Phase: Unterste Reihe des Hofs befreien
2. Phase: Blasen des gesamten Laubs in die oberste Reihe
3. Phase: Konzentrieren des gesamten Laubs der obersten Reihe auf dem Randfeld, das sich in der selben Spalte wie Q befindet.
4. Phase: Transferieren des auf Feld $(Q[0], 0)$ gesammelten Laubs auf Feld Q.

Für jede dieser Phasen wird ein Algorithmus entwickelt, der die während der Phase auszuführenden Blasoperationen und Muster erstellt und zum generalisierten Ablaufplan hinzufügt.

Rotation des Hofs: Die Algorithmen zur Erstellung des generalisierten Ablaufplans sind am effektivsten, wenn sich das Randfeld Feld, auf dem das Laub in der 3. Phase konzentriert wird, möglichst nahe am Feld Q befindet. Dies liegt daran, dass Phase 4 sehr „kostenintensiv“ (im Sinne von viele Blasoperationen erfordernd ist), wenn Feld Q weit von dem Feld entfernt ist, auf dem in der 3. Phase das Laub konzentriert wurde (mehr dazu später). Zur Vermeidung unnötiger Blasoperationen muss dies berücksichtigt werden.

Um die Algorithmen zur Erstellung des Ablaufplans zu vereinfachen, wird vor ihrer Durchführung der Hof mitsamt Feld Q solange um 90° gegen den Uhrzeigersinn rotiert, bis die Kante, die am nächsten an Feld Q ist, die obere Kante des Hofs ist. Dies hat zur Folge, dass das Laub nun immer auf dem Feld $(Q[0], 0)$ (das Feld am oberen Rand, das sich am nächsten an Q befindet) konzentriert werden kann und sich die Algorithmen nicht dynamisch an die Kante anpassen müssen, die am nächsten an Q ist.

Diese Rotation wird im Folgenden als „Initialrotation“ bezeichnet, die Anzahl an durchgeföhrten

90°-Rotationen wird gespeichert.

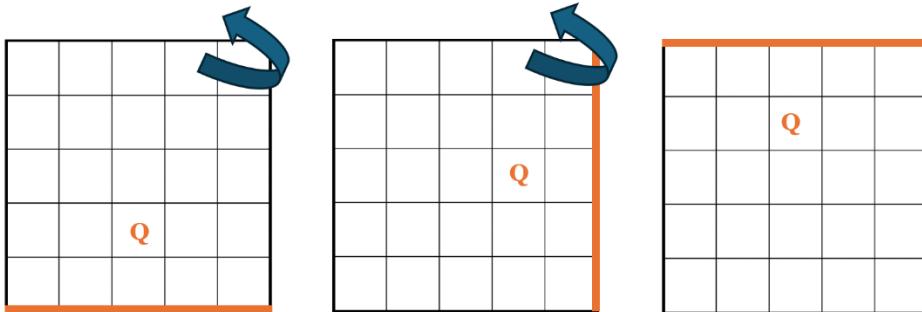


Abbildung 16: Veranschaulichung der Initialrotation: Der Hof wird solange um 90° gegen den Uhrzeigersinn gedreht, bis die Kante, die Feld Q am nächsten ist (rot markiert), die obere Kante ist bzw. bis Feld Q in der oberen Hälfte des Kreises ist. Die Algorithmen zum Erstellen des Ablaufplans werden anschließend auf den rotierten Hof angewendet.

Wenn nun der Hof rotiert wurde und anschließend ein Muster oder eine Blasoperation zum generalisierten Ablaufplan hinzugefügt wird, dann muss immer zunächst die Initialrotation rückgängig gemacht werden, d.h. Feld 0 und Blasrichtung der jeweiligen Blasoperation müssen mit dem Uhrzeigersinn rotiert werden.

Die Rotation eines sich auf dem Feld befindenden Punkts um 90° gegen den Uhrzeigersinn kann mit folgender Formel berechnet werden, die von der alten Position des Punkts und der Seitenlänge n des Hofs abhängt:

$$P_{rotated90ccw} = (P[1], n - 1 - P[0])$$

Eine Rotation um 90° kann ausgedrückt werden, indem die Umkehrung der obigen Formel gebildet wird:

$$P_{rotated90cw} = (n - 1 - P[1], P[0])$$

Im folgenden wird für jede Phase des Conquer-Verfahrens ein Algorithmus vorgestellt, der die in der Phase auszuführenden Blasoperationen und Muster ermittelt. Die Algorithmen arbeiten dabei unter der Voraussetzung, dass sich Feld Q in der oberen Hälfte des Hofs befindet. Diese Voraussetzung ist über die Initialrotation sichergestellt.

1. Phase: Unterste Reihe des Hofs befreien

In der ersten Phase wird die unterste Reihe des Hofs so vollständig wie möglich geleert. Zunächst wird das Laub der Nicht-Eckfelder, die sich in der untersten Reihe befinden, mit einem nachhaltigen Verfahren auf das Eckfeld in der Ecke links unten geblasen, sodass die Nicht-Eckfelder in der untersten Reihe vollständig geleert sind.

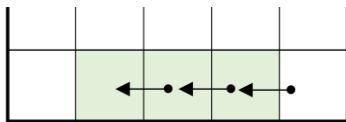


Abbildung 17: Die Blasoperationen, die hierbei ausgeführt, am Beispiel eines Hofs der Seitenlänge 5. Die eingezeichneten Blasoperationen werden von rechts nach links ausgeführt. Dabei werden die grün eingefärbten Felder vollständig geleert.

Anschließend werden zwei Muster ausgeführt, die die beiden Eckfelder in der untersten Reihe asymptotisch leeren und das Laub der Eckfelder dabei auf die Randfelder in der zweit-untersten Reihe transferieren.

100	100	100	100	100	100
100	100	100	100	100	100
100	100	100	100	100	100
100	100	100	100	100	100
557	120	112	108	100	196
3	0	0	0	0	4

Abbildung 18: Beispiel - ein Hof der Seitenlänge 6 nach Durchführen der 1. Phase. Die unterste Reihe ist bis auf die Eckfelder vollständig geleert.

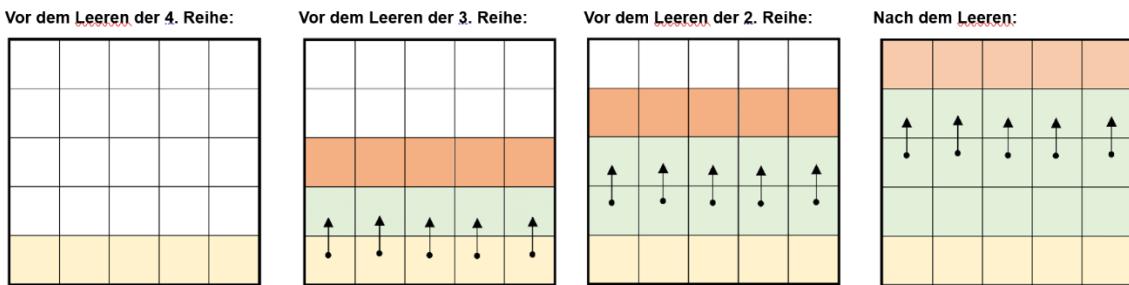
Algorithmus 10: clear_bottom_line bzw. Algorithmus zum Hinzufügen der Blasoperationen und Muster von Phase 1 zum generalisierten Ablaufplan

Globale Variable: strategy = Liste, die den generalisierten Ablaufplan speichert

Parameter: n = Seitenlänge des Hofs

1. für jedes start_x im Bereich [n-1,0] (Änderung von start_x pro Iteration: -1):
 2. füge Blasoperation, die vom Feld (start_x, n-1) in Richtung (-1,0) bläst zu strategy hinzu
 3. füge Muster, das das Laub vom Eckfeld (0, n-1) auf das Randfeld (0, n-2) transferiert zu strategy hinzu
 4. füge Muster, das das Laub vom Eckfeld (n-1,n-1) auf das Randfeld (n-1, n-2) transferiert zu strategy hinzu
2. Phase: Laub auf oberste Reihe bringen

In der zweiten Phase wird das gesamte Laub auf die oberste Reihe gebracht. Hierfür wird von unten nach oben Reihe für Reihe vollständig geleert. Das Vorgehen, das bereits in einem Vorkapitel erläutert wurde (auf eine Formalisierung als Pseudocode wird daher verzichtet), wird in der folgenden Grafik an einem Hof der Dimensionen (5,5) veranschaulicht:



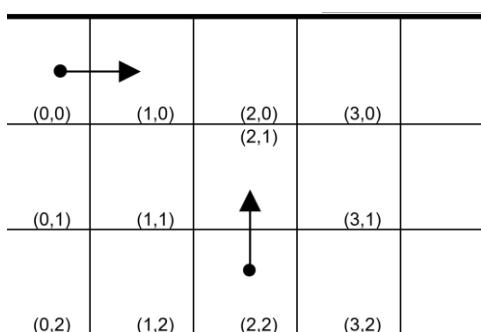
3. Phase: Laub auf der obersten Reihe konzentrieren

In der 3. Phase wird das sich auf der oberen Reihe befindende Laub auf den Feldern $(Q[0],0)$, $(Q[0]-1,0)$ und $(Q[0]+1,0)$ versammelt.

Zunächst werden hierfür die Eckfelder $(0,0)$ und $(n-1,0)$ über Muster geleert, das Laub wird dabei auf die Nachbarfelder der Eckfelder, die sich auf der obersten Reihe befinden, geblasen.

Anschließend wird das Laub, das sich auf der obersten Reihe befindet, auf Feld $(Q[0],0)$ geblasen. Durch die Anwendung von Mustern wird versucht, die entstehenden dabei Seitenabtriebe (die auf Felder in der Reihe unter der obersten Reihe fliegen) zu eliminieren bzw. die von den Seitenabtrieben betroffenen Felder asymptotisch zu leeren.

Folgende Grafik veranschaulicht ein Beispiel, in dem ein Muster verwendet wird, das das Feld $(1,0)$ asymptotisch leert, gleichzeitig das Feld $(2,1)$, auf das bei der Blasoperation $\text{blase}((1,0),(1,0))$ Seitenabtriebe gelangen, ebenfalls asymptotisch leert und die Laubmenge auf Feld $(2,0)$ kontinuierlich erhöht:



Wenn die schwarz eingezeichneten Blasoperationen sehr oft durchgeführt werden, dann werden die Felder $(1,0)$ und $(2,1)$ asymptotisch geleert. Nach Beendigung des Musters kann anschließend das Feld $(2,0)$ mit einem neuen Muster, das auf dem gleichen Prinzip basiert, geleert werden und dabei das Laub auf das Feld $(3,0)$ geschafft werden usw., bis das gesamte Laub der oberen Reihe auf den Feldern $(Q[0],0)$, $(Q[0]-1,0)$ und $(Q[0]+1,0)$ versammelt ist.

Algorithmus 11: concentrate_top_line bzw. Algorithmus zum Hinzufügen der Blasoperationen und Muster von Phase 3 zum generalisierten Ablaufplan

Globale Variable: strategy = Liste, die den generalisierten Ablaufplan speichert

Parameter: n = Seitenlänge des Hofs

- source_fields_for_edgeclear = $[(Q[0]-1,0), (Q[0],0), (Q[0]+1,0)]$ (es handelt sich hierbei um die Randfelder, auf denen das Laub versammelt werden soll)

2. **falls** (0,0) nicht in der Liste source_fields_for_edgeclear: (wenn das Laub auf einem dieser Randfelder versammelt werden soll, dann macht es keinen Sinn, es von ebendiesem Randfeld herunterzublasen)
3. **füge** Muster, das das Laub vom Eckfeld (0,0) auf das Randfeld (1,0) transferiert **zu** strategy **hinz**
4. **falls** (n-1,0) nicht in der Liste source_fields_for_edgeclear:
5. **füge** Muster, das das Laub vom Eckfeld (n-1,0) auf das Randfeld (n-2,0) transferiert **zu** strategy **hinz**
6. **für jedes** start_x **im Bereich** [0;n-1] (Änderung von start_x pro Iteraton: 1):
 7. **falls** (start_x+1, 0) in der Liste source_fields_for_edgeclear:
 8. **Schleife abbrechen**
 9. **andernfalls:**
 10. m = Muster, das nach dem oben erläuterten Prinzip das Laub vom Randfeld (start_x+1,0) auf das Randfeld (start_x+2,0) schafft und dabei die Seitenabtriebe eliminiert
 11. **füge** m **zu** strategy **hinz**
 12. **für jedes** start_x **im Bereich** [n-1;0] (Änderung von start_x pro Iteraton: -1):
 13. **falls** (start_x-1, 0) in der Liste source_fields_for_edgeclear:
 14. **Schleife abbrechen**
 15. **andernfalls:**
 16. m = Muster, das nach dem oben erläuterten Prinzip das Laub vom Randfeld (start_x-1,0) auf das Randfeld (start_x-2,0) schafft und dabei die Seitenabtriebe eliminiert
 17. **füge** m **zu** strategy **hinz**

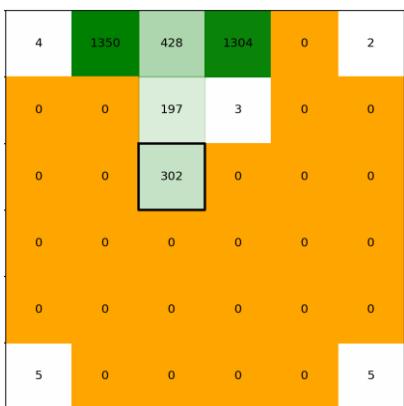


Abbildung 19: Ein Beispielhof nach Vollendung von Phase 3

4. Phase: Gesamtes Laub auf Feld Q verschieben

Wenn Q ein Nachbarfeld eines Randfelds ist, dann kann über das Muster zum Transferieren des Laubs eines Randfelds auf ein Nicht-Randfeld die Laubmenge auf Q asymptotisch maximiert bzw. die Laubmenge des benachbarten Randfelds asymptotisch minimiert werden.

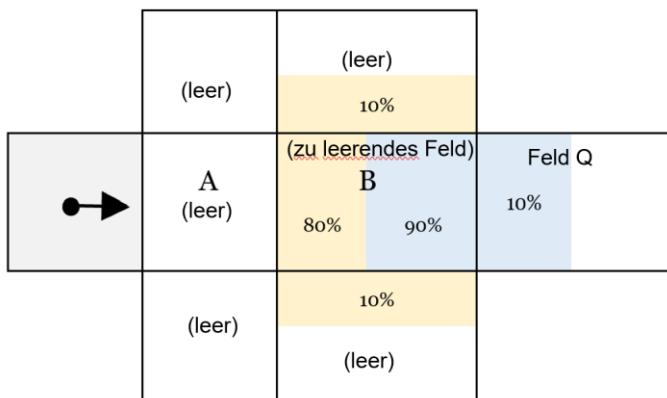
Dies geht aber nur, wenn Q Nachbar eines Randfelds ist – für andere Fälle muss ein anderes Verfahren entwickelt werden.

Warum ist es nicht sinnvoll, direkt auf Q zu blasen?

Beim direkten Blasen auf Q (sodass Feld Q Feld B der Blasoperation ist und das Nachbarfeld von Feld Q, dessen Laub auf Q geblasen werden soll, Feld A der Blasoperation ist) gelangt zwar Laub auf Q, allerdings ...

- gelangt nicht das gesamte Laub von Feld A auf Q, da ca. 20% des Laubs von Feld A seitlich abgetrieben wird
- werden außerdem 10% des Laubs von Feld Q nach vorne abgetrieben. Wenn sich bereits viel Laub auf Q befindet, kann es sein, dass der Zusammenhang
 $10\% * \text{Blattanzahl_auf_Q} > 80\% \text{Blattanzahl_auf_A}$
erfüllt ist. In diesem Fall hätte ein direktes Blasen auf Feld Q also sogar zur Folge, dass sich die Blattanzahl auf Feld Q reduzieren würde. Wenn man nur durch direktes Blasen auf Feld Q Laub auf Q befördert, dann kann man folglich die Laubmenge auf Q nicht asymptotisch maximieren bzw. die Laubmengen auf den anderen Feldern nicht asymptotisch minimieren, da sich diese Laubmengen durch die zuvor erläuterten Abtriebe nicht Null annähern werden.

Sinnvoller ist es, das Laub so auf Feld Q zu blasen, dass das zu leerende Feld mit Feld B der Blasoperation korrespondiert (im Folgenden als „Verfahren zum abtriebsfreien Transportieren“ bezeichnet):



Hierbei hat man keine Seitenabtriebe, und es gelangen nur die 10% des Laubs, die auf Feld B nach vorne abgetrieben werden, auf Feld Q. Die Blasoperation muss dabei sofort durchgeführt werden, bis die Laubmenge auf B kleiner als *tolerated_amount* ist – dies kann über ein Muster umgesetzt werden.

Problem: Dies ist nur möglich, wenn sich unter Feld Q drei weitere Felder befinden, da nur dann ein Feld 0 existiert, von dem aus so wie in der Grafik dargestellt geblasen werden kann. Auf einem Hof der Maße (5,5) oder (6,6) und Q=(2,2) ist dem nicht der Fall – daher handelt es sich bei diesem Hof um einen Sonderfall, der gesondert zu betrachten ist.

Bauen des generalisierten Ablaufplans der 4. Phase

Zunächst wird mit einem Muster das Laub vom Randfeld ($Q[0],0$) auf das Nicht-Randfeld ($Q[0],1$) gebracht.

Das Laub ist nach diesem Schritt auf dem Feld ($Q[0],1$) gesammelt - wenn $Q[1] == 1$, dann hat man also sein Ziel erreicht und der Algorithmus kann beendet werden.

Transferieren des Laubs vom Feld ($Q[0],1$) auf das Feld ($Q[0],2$):

Wenn $Q[1] \neq 1$, dann ist man aber leider noch nicht fertig. Da $Q[1] \neq 1$ gilt nun zwangsläufig $Q[1] > 1$, d.h. das Laub muss in eine tiefere Zeile gebracht werden. Um das Laub vom Feld $(Q[0],1)$ auf $(Q[0],2)$ zu bringen und dabei $(Q[0],1)$ vollständig und alle anderen beteiligten Felder (außer $Q[0],2$) asymptotisch zu leeren, kann wie folgt vorgegangen werden:

1. Zuerst stellt sich der Hausmeister auf Feld $(Q[0],0)$ und bläst in die Richtung $(0,1)$ bzw. nach unten. Hierdurch wird das Feld $(Q[0],1)$ vollständig geleert und ein Großteil des Laubs gelangt auf $(Q[0],2)$, es gelangen aber auch Seitenabtriebe auf die Felder $(Q[0],3)$, $(Q[0]+1,2)$ und $(Q[0]-1,2)$.

Schritt 2 und Schritt 3 werden nur dann ausgeführt, wenn $n \neq 5$ ist (im Sonderfall $n = 5$ wird anders vorgegangen):

2. Das Laub vom Feld $(Q[0]+1,2)$ muss nun seitenabtriebsfrei auf $(Q[0]+1,3)$ geblasen werden und $(Q[0]+1,2)$ muss dabei asymptotisch geleert werden. Hierfür wird ein Muster verwendet, das die Blasoperation $\text{blase}((Q[0]+1,0), (0,1))$ solange ausführt, bis die Laubmenge auf Feld $(Q[0]+1,2)$ kleiner als tolerated_amount ist.
3. Das Laub vom Feld $(Q[0]-1,2)$ muss nun seitenabtriebsfrei auf $(Q[0]-1,3)$ geblasen werden und $(Q[0]-1,2)$ muss dabei asymptotisch geleert werden. Hierfür wird ein Muster verwendet, das die Blasoperation $\text{blase}((Q[0]-1,0), (0,1))$ solange ausführt, bis die Laubmenge auf Feld $(Q[0]-1,2)$ kleiner als tolerated_amount ist.

Nach Schritt 3 sieht die Blattverteilung auf dem Hof wie folgt aus:

	$(Q[0]-1,0)$	$(Q[0],0)$	$(Q[0]+1,0)$	
	$(Q[0]-1,1)$	$(Q[0],1)$	$(Q[0]+1,1)$	
	$(Q[0]-1,2)$	$(Q[0],2)$	$(Q[0]+1,2)$	
	$(Q[0]-1,3)$	$(Q[0],3)$	$(Q[0]+1,3)$	
	$(Q[0]-1,4)$	$(Q[0],4)$	$(Q[0]+1,4)$	

Abbildung 20: Die hellrot markierten Felder sind asymptotisch geleert worden, auf den braunen Feldern befindet sich viel Laub

Für die nächsten Schritte muss eine Fallunterscheidung vorgenommen werden.

Fall $n \geq 7$:

In diesem Fall sind unter $(Q[0],2)$, noch mind. vier weitere Felder. Daher kann folgendes Vorgehen verwendet werden, um das gesamte Laub auf Feld $(Q[0],2)$ zu bringen:

1. Die Blasoperationen $\text{blase}((Q[0]-2,3), (1,0))$ und $\text{blase}((Q[0]-2,3), (1,0))$ eingezeichnet. Nach den beiden Blasoperationen sieht die Blattverteilung folgendermaßen aus (die beiden Blasoperationen sind ebenfalls in dem Hofausschnitt eingezeichnet):

	(Q[0]-1,0)	(Q[0],0)	(Q[0]+1,0)	
	(Q[0]-1,1)	(Q[0],1)	(Q[0]+1,1)	
	(Q[0]-1,2)	(Q[0],2)	(Q[0]+1,2)	
	(Q[0]-1,3) (vollst. leer)	(Q[0],3)	(Q[0]+1,3) (vollst. leer)	
	(Q[0]-1,4)	(Q[0],4)	(Q[0]+1,4)	

2. Mit dem zuvor erläuterten Verfahren zum abtriebsfreien Transportieren wird zunächst das Laub von Feld (Q[0],4) auf (Q[0],3) befördert, indem die Blasoperation blase((Q[0], 6), (0,-1)) so oft wiederholt wird, bis die Blattmenge auf (Q[0],4) kleiner als tolerated_amount ist.
3. Anschließend wird das Laub von Feld (Q[0],3) auf (Q[0],2) befördert, indem die Blasoperation blase((Q[0], 5), (0,-1)) so oft wiederholt wird, bis die Blattmenge auf (Q[0],3) kleiner als tolerated_amount ist.

Nach diesen Schritten befindet sich das Laub auf (Q[0],2) - wenn Q[1] == 2, dann hat man also sein Ziel erreicht und der Algorithmus kann beendet werden. Wenn nicht, dann muss das Laub noch von (Q[0],2) auf Feld Q transportiert werden – dies kann mit dem Verfahren zum abtriebsfreien Transportieren seitenverlustfrei durchgeführt werden.

Fall n = 6:

In diesem Fall kann die Blasoperation blase((Q[0], 6), (0,-1)) nicht durchgeführt werden, da die das Feld (Q[0],6) nicht im Hof existiert. Stattdessen muss ein anderes Verfahren verwendet werden, dass leider mehr Blasoperationen erfordert als das im Fall n = 7 verwendete Verfahren.

Es wird folgendes Muster ausgeführt, um das gesamte Laub auf Q zu bringen - die Source-Felder des Musters sind dabei (Q[0], 3), (Q[0], 4), (Q[0]-1, 3) und (Q[0]+1, 3):

- | |
|-----------------------------|
| 1. blase((Q[0]-2,3),(1,0)) |
| 2. blase((Q[0]+2,3),(-1,0)) |
| 3. blase((Q[0],5),(0,-1)) |

Nach dem Durchführen dieses Musters befindet sich das Laub auf (Q[0],2) - wenn Q[1] == 2, dann hat man also sein Ziel erreicht und der Algorithmus kann beendet werden. Wenn nicht, dann muss das Laub noch von (Q[0],2) auf Feld Q transportiert werden – dies kann mit dem Verfahren zum abtriebsfreien Transportieren seitenverlustfrei durchgeführt werden.

Fall n = 5:

Der Sonderfall $n = 5$ ist am schwierigsten zu lösen bzw. man benötigt hier am meisten Blasoperationen, um die Laubmenge auf Q asymptotisch zu maximieren und die anderen Felder asymptotisch zu leeren. Im Fall $n = 5$ ist Q auf jeden Fall $(2,2)$, da alle anderen möglichen Positionen von Q Nachbarn von Randfeldern sind - und in diesem Fall wäre das Laub schon längst auf Q gebracht worden (zur Erinnerung siehe Seite 37: „Das Laub ist nach diesem Schritt auf dem Feld $(Q[0],1)$ gesammelt - wenn $Q[1] == 1$, dann hat man also sein Ziel erreicht und der Algorithmus kann beendet werden.“)

Es wird zunächst die Blasoperation $\text{blase}((2,4),(0,-1))$ ausgeführt. Diese Blasoperation bläst Laub direkt auf Feld Q . Dies soll ja eigentlich vermieden werden – für einen Hof mit den Seitenlängen $(5,5)$ und $Q=(2,2)$ wurde durch Testen bzw. Ausprobieren herausgefunden, dass diese Blasoperation die Laubmenge auf Q erhöht.

Anschließend wird mit dem Verfahren zum abtriebsfreien Transportieren das Laub von den Feldern $(Q[0]-1,2)$ und $(Q[0]+1,2)$ auf die Felder $(Q[0]-1,1)$ und $(Q[0]+1,1)$. Der Grund dafür, dass im Sonderfall $n = 5$ die dritte, sondern die erste Reihe zum Sammeln des Laubs verwendet wird: Das im nächsten Schritt definierte Muster, das die Laubmenge auf Q asymptotisch maximiert, geht mit sehr vielen Feldern einher, die nur asymptotisch geleert werden können. Ziel des 2. Ansatzes ist es, die Anzahl an Feldern, die nur asymptotisch und nicht vollständig geleert werden können, möglichst gering zu halten. Daher ist es sinnvoll, für dieses Muster die Felder $(Q[0]-1, 0)$, $(Q[0],0)$ und $(Q[0]+1,0)$ zu verwenden, die ohnehin schon zu Beginn der 4. Phase nur asymptotisch geleert werden konnten und daher quasi „vorbelastet“ sind, anstatt die Felder in der unterstehen Reihe zu verwenden, die alle vollständig geleert sind und „neu belastet würden“, würde das Muster auf diesen Feldern ausgeführt werden.

Dieses Muster, das als nächstes auszuführen ist, hat die Source-Felder $(1,1)$, $(2,1)$, $(3,1)$, $(2,0)$ und beinhaltet folgende Blasoperationen:

1. Muster, das die Source-Felder $(1,1),(2,1),(3,1)$ hat und folgende Blasoperationen beinhaltet:
 1. $\text{blase}((0,1),(1,0))$
 2. $\text{blase}((4,1),(-1,0))$
2. Muster, das das Laub vom Randfeld $(2,0)$ auf das Nicht-Randfeld $(2,1)$ transferiert

Das im Sonderfall $n = 5$ durchgeführte Muster ähnelt konzeptionell dem im Sonderfall $n = 6$ durchgeführten Muster, es ist aber an den kleineren Hof angepasst.

Anzahl durchzuführender Blasoperationen – Korrelation zu Hofgröße:

Es fällt auf: Je kleiner der Hof, desto komplizierter ist die Durchführung von Phase 4 bzw. desto mehr Blasoperationen müssen durchgeführt werden. Dies mag einem antiintuitiv vorkommen. Ursächlich für den Umstand ist, dass auf einem großen Hof das Verfahren zum abtriebsfreien Transportieren von Laub auf Q uneingeschränkt funktioniert, was auf einem kleinen Hof nicht mehr der Fall ist, da dort die entsprechenden Felder außerhalb des Hofs liegen,

Optimierungen

Im den vorherigen Kapiteln zum 2. Ansatz wurden einige umgesetzte Optimierungen noch nicht angesprochen. Diese werden im Folgenden aufgelistet:

- *Muster-Konzept*: Wenn während der Ausführung eines Musters festgestellt wird, dass sich die Gesamtblaubmenge auf den Source-Feldern nicht mehr ändert und die Erwartungswerte zur Wahrscheinlichkeitsmodellierung angewendet werden, dann wird das Muster frühzeitig abgebrochen – auf die Art werden unnötige Blasoperationen, die eh nichts mehr bringen, vermieden. Dieser Fall kann aber nur auftreten, wenn irgendwo anders in der Implementierung ein Bug ist.
- *Phase 4 mit Sonderfall n = 5*: Die Durchführung von Phase 4 geht schneller, wenn das Muster zum Maximieren der Laubmenge auf Q von einer anderen Kante aus durchgeführt wird – dies hat aber eine Verringerung der Anzahl an vollständig geleerten Feldern zur Folge. Über den Parameter `choose_faster_path` kann eingestellt werden, wie in dem Fall verfahren wird – ist `choose_faster_path` wahr, dann wird die Geschwindigkeit priorisiert, ansonsten wird die Maximierung der Anzahl vollständig geleerter Felder priorisiert.

Gesamt-Algorithmus und Kritik

Der Gesamt-Algorithmus nimmt folgende Parameter:

- `Q` (Tupel): Ein Tupel, das den Index von Feld Q angibt
- `hof_size` (Tupel): Ein Tupel, das die Größe des Hofs angibt
- `startwert` (ganze Zahl): Gibt die Anzahl an Blättern an, die sich zu Beginn auf den Feldern befindet
- `use_binomial` (Wahrheitswert): Gibt an, ob bei der Simulation der Blasvorgänge am tatsächlichen Hof in Algorithmus 7 die Erwartungswerte verwendet werden sollen oder ob die tatsächlichen Wahrscheinlichkeiten zu simulieren sind
- `tolerated_amount` (ganze oder rationale Zahl): in den Vorkapiteln erläutert
- `max_muster_operations` (ganze Zahl): Anzahl an Operationen, die pro Muster maximal durchgeführt werden (ein Muster wird auf jeden Fall abgebrochen, wenn es `max_muster_operations` Blasoperationen durchgeführt hat)
- `choose_faster_path` (Wahrheitswert): Im Vorkapitel erläutert

Der zweite Ansatz ist aus mehreren Gründen deutlich positiver zu bewerten als der erste Ansatz:

- Deutlich bessere Ergebnisse: Das Ziel des Hausmeisters, so viel Laub wie möglich auf Q zu versammeln lässt sich mit dem 2. Ansatz optimal erreichen, da er dafür sorgt, dass auf allen Feldern außer Q die Laubmenge gegen Null konvergiert (Voraussetzung: es werden die deterministischen Erwartungswerte zur Wahrscheinlichkeitsmodellierung verwendet⁵). Es lässt sich also sagen, dass der zweite Ansatz die Aufgabenstellung optimal löst – dies ist beim 1. Ansatz nicht der Fall.

⁵ Bei Simulation des Zufalls würde eine Aussage wie „die Blattmenge konvergiert gegen ...“ gar keinen Sinn machen.

- Der Hausmeister kann über den Parameter tolerated_amount steuern, wie viele Blätter er auf einem nur asymptotisch leerbaren Feld toleriert.
- Deutlich systematisches Vorgehen als beim ersten Ansatz.
- Die Laufzeit ist auch deutlich besser (siehe nächstes Kapitel)

Einzig und allein der Umstand, dass der generalisierte Programmablaufplan zu Teilen nicht dynamisch ist und diese nicht-dynamischen Teile bereits vor Beginn des Blasprozesses endgültig festgelegt werden, ist möglicherweise etwas kontrovers zu betrachten, denn in der Aufgabe heißt es explizit: „Der Hausmeister soll sich vor jedem Blasvorgang entscheiden ...“, dies legt ein dynamisches Verfahren nahe.

Die Aufgabenstellung verbietet allerdings nicht, dass der Hausmeister ein Gedächtnis hat, und wie zuvor gezeigt handelt es sich bei der in Ansatz 2 verwendeten Vorgehensweise um eine Vorgehensweise, die das Problem optimal löst. Außerdem kommen in Ansatz 2 ja durchaus dynamische Teile vor (Muster). Von daher halte ich den zweiten Ansatz für den besten Ansatz bzw. **der zweite Ansatz, der zuvor erläutert wurde und in der Datei aufgabe1_solver2.py implementiert ist, soll das zu bewertende Verfahren sein.** Die anderen Verfahren dienen als Vergleichsgegenstände bzw. als andere mögliche Verfahren, die sich aber als schlechter erwiesen haben.

Laufzeit

Die Laufzeiten der einzelnen Bestandteile des 2. Lösungsverfahren:

1. Laufzeit des Verfahrens, das die in Phase 1 auszuführenden Operationen ermittelt: $O(n)$, da einmal über alle Felder der untersten Reihe iteriert wird
2. Laufzeit des Verfahrens, das die in Phase 2 auszuführenden Operationen ermittelt: $O(n^2 - n) = O(n^2)$, da einmal über alle Felder bis auf die Felder der obersten Reihe iteriert wird
3. Laufzeit des Verfahrens, das die in Phase 3 auszuführenden Operationen ermittelt: $O(n)$
4. Laufzeit des Verfahrens, das die in Phase 4 auszuführenden Operationen ermittelt: Schwer zu quantifizieren. Wenn alle Parameter außer n als konstant angenommen werden, dann ist die Laufzeit $O(1)$. Wenn alle Parameter außer n und Q als konstant angenommen werden, dann ist die Laufzeit $O(\max(0, Q[1]-2))$ (wegen dem Programmteil, der das Laub vom Feld $(Q[0], 2)$ auf das Feld $(Q[0], 3)$ transferiert).
5. Laufzeit des Verfahrens, was die Operationen des generalisierten Ablaufplans ausführt: $O(m)$ mit $m :=$ Anzahl an Operationen, die durchgeführt werden müssen, um den generalisierten Ablaufplan einmal zu durchlaufen. m ist jedoch schwer zu generalisieren und außerdem vom Zufall abhängig, wenn der tatsächliche Zufall simuliert wird.

Die Gesamlaufzeit wird dominiert durch $O(n^2)$. Sie ist deutlich besser als die Gesamlaufzeit des 1. Verfahrens, die $O(n^2 * \text{max_operation})$ beträgt. Anders ausgedrückt: In derselben (theoretischen) Zeit, in der der 1. Ansatz eine einzige Blasoperation simuliert, kann der 2. Ansatz einmal ganz durchlaufen.

Kombination der Verfahren (3. Ansatz)

Es stellt sich die Frage, ob der 1. Ansatz verbessert werden kann, indem Teile des 2. Ansatz in ihn eingebunden werden. Es wird also ein 3. Ansatz entwickelt, der auf dem 1. Ansatz aufbaut und konkret folgende Bestandteile des 2. Ansatz in den 1. Ansatz einbindet:

- Die Muster zum Leeren von Eck- und Randfeldern
- Das Muster aus Phase 4 des 2. Ansatz (Transferieren des Laubs auf Q).

Als der dritte Ansatz entwickelt wurde, bestand die Hoffnung, durch die Kombination ein Verfahren schaffen zu können, das sowohl über die Steuerbarkeit des 1. Ansatzes als auch über die systematische Herangehensweise des 2. Ansatzes verfügt. Die Ergebnisse zeigen jedoch auf, dass dies nicht wirklich gelungen ist.

Der 3. Ansatz ist kaum optimiert und auf keinen Fall das Verfahren, das bewertet werden soll. Es dient vielmehr als Vergleichsgegenstand zum 2. Ansatz (dem besten Ansatz).

Parametrisierung

Durch die Kombination des 1. und 2. Ansatzes verfügt der 3. Ansatz über alle Parameter, über die der 1. Ansatz und der 2. Ansatz verfügen. Parameter sind grundsätzlich gut, da sie dem Nutzer Anpassungsmöglichkeiten geben. Bei einer zu großen Anzahl an Parametern ist allerdings fraglich, ob es wirklich sinnvoll, so viele Parameter zu verwenden – insbesondere dann, wenn eine Veränderung verschiedener Parameter ähnliche Auswirkungen auf das Resultat hat.

Zusammenfassung

Es wurden drei Ansätze entwickelt. Ansatz 1 verwendet dabei eine Greedy-Heuristik, deren Heuristik eine Kombination zweier Heuristik-Kenngrößen ist, die über Parameter gewichtet werden können.

Ansatz 2 verwendet einen systematischen Conquer-Algorithmus, um möglichst viele Felder vollständig zu leeren und alle anderen Felder (außer Q) asymptotisch zu leeren. Dieser Ansatz liefert die besten Ergebnisse und ist der zu bewertende Ansatz.

Ansatz 3 versucht, Ansatz 1 und 2 zu kombinieren, um die Vorteile beider Ansätze auszunutzen, was aber nur bedingt gelingt und zu einem Parameterchaos führt.

Bereits in Solver 1 bis 3 enthaltene Erweiterungen

- Ansatz 1 und 3 lassen sich problemlos auch auf Höfe anwenden, die nicht quadratisch, sondern rechteckig sind. Diese Erweiterung wird direkt in dem Programm, das die Grundaufgabe löst, implementiert, da mit ihr keine Komplexitätszunahme einhergeht.
- An Ansatz 2 müssen einige kleine Änderungen vorgenommen werden, damit er anwendbar auf rechteckige Höfe wird: Die Initialrotation muss bei einem Hof mit den Maßen (3,x) mit $x > 3$ so lange durchgeführt werden, bis die obere Kante die Länge x hat. Die obere Kante darf nämlich nicht die Länge 3 haben, da am oberen Rand das Verfahren zum Transferieren des Randlaubs auf ein Nicht-Randlaub durchgeführt wird, und dieses Verfahren funktioniert nur an Randsegmenten, die länger als 4 Felder sind.

- Aus der Aufgabe geht nicht klar hervor, ob erwartet wird, dass ein Feld Q als Parameter gegeben wird, oder ob das Programm automatisch das am besten geeignete Feld Q auswählen soll. Die Programme sind auf jeden Fall so konzipiert, dass Q als Parameter mitgegeben wird, und können mit allen möglichen Q umgehen.

Wenn zum Durchführen des Laubblasprozesses Ansatz 2 verwendet wird, dann sind für Q Felder geeignet, die Nachbarn von Randfeldern sind und die zum nächsten Eckfeld eine Manhattan-Distanz größer als 2 haben. Grund hierfür: Der Prozess, mit dem das Laub in Phase 4 von $(Q[0],1)$ auf $(Q[0],2)$ befördert wird, ist in Hinblick auf die benötigten Blasoperationen gerade bei Höfen mit $n = 5$ oder $n = 6$, aber auch bei anderen Hofgrößen aufwändig. Bei Feldern, die Nachbarn von Randfeldern sind, muss dieser Prozess nicht durchgeführt werden.

Erweiterung 1: Keine Mauer-Umrandung des Hofs

Diese Erweiterung basiert auf dem 2. Ansatz.

Zunächst müssen die Blasregeln an die neuen Gegebenheiten angepasst werden: Wenn der Hof kein Mauer mehr hat, dann ist es möglich, dass Laub den Hof verlässt.

Die zu Beginn des Dokuments beschriebenen Regeln legen in Sonderfällen normalerweise fest, dass das Laub, das überlicherweise auf ein nicht existierendes Feld geflogen wäre, nun stattdessen vollständig auf dem Feld neben der Mauer verbleibt. Diese Festlegung ergibt keinen Sinn mehr, wenn der Hof nicht mehr von einer Mauer umgeben ist. „Kein Mauer“ bedeutet aber auch nicht zwangsläufig, dass die Umrandung des Hofs komplett offen ist: Was, wenn der Hof von einem Maschendrahtzaun umgeben ist, durch den zwar manche Blätter durchfliegen können, der aber immer noch einen gewissen Widerstand auf die seitlich abtreibenden Blätter ausübt?

Wenn Laub gegen die Umrandung geblasen oder an der Umrandung entlanggeblasen wird, dann wird zumindest der dabei entstehende Seitenabtrieb auf jeden Fall mit der Umrandung in Berührung kommen. Es wird der Parameter *wall_resistance* eingeführt, der angibt, wie groß der Anteil der Blätter, die den Hof nicht verlassen, an den Blättern ist, die insgesamt mit der Umrandung in Berührung kommen. Der Parameter kann durch Kalibrieren an die entsprechenden örtlichen Gegebenheiten angepasst werden. Im Folgenden wird mit *wall_resistance* = 0,5 gearbeitet (das heißt, 50% der Blätter, die gegen die Umrandung fliegen, verlassen den Hof – die anderen 50% verbleiben im Hof und fliegen auf das Feld, auf das sie den regulären Blasregeln nach fliegen würden).

Nach Anpassung der Blasregeln gelangt man schnell zu der Feststellung, dass auch die Algorithmen, die in den einzelnen Phasen durchzuführenden Operationen zum generalisierten Ablaufplan hinzufügen, angepasst werden müssen. Es ist jetzt, wo der auf die Umrandung treffende Seitenabtrieb teilweise verloren geht, definitiv keine gute Idee mehr, Laub unnötigerweise am Rand entlangzublasen. Das Ziel des Hausmeisters ist schließlich klar definiert als „möglichst viel Laub auf Q bringen. Laub, das einmal den Hof verlassen hat, kann nie wieder auf ihn zurückgebracht werden und wirkt sich daher negativ auf das Erreichen des Ziels des Hausmeisters aus.“

Es müssen also alle Phasen so abgeändert werden, dass möglichst wenig Operationen durchgeführt werden, die Laub am Rand entlangblasen – hierbei handelt es sich nun um das primäre Ziel. Das

erste Ziel des 2. Ansatzes (möglichst viele Felder vollständig leeren) ist jetzt erst mal sekundär, denn noch nicht vollständig geleerte Felder können später immer noch geleert werden, aber Laub, das den Hof einmal verlassen hat, kann nie wieder zurückgewonnen werden. Das tertiäre Ziel, die Anzahl an Blasoperationen zu verringern, rückt dabei nun vollständig in den Hintergrund.

Hierfür werden die Phasen wie folgt abgeändert: (die neue Funktionsweise wird an einem Beispielhof visualisiert, der die Dimensionen (10,10) hat).

Abgeänderte 1. Phase (jetzt: Ränder (bis auf den oberen Rand) entlauben):

In der 1. Phase werden nun zunächst die Blätter, die sich auf dem unteren Rand befinden und keine Eckfelder sind, alle auf die Felder (1,n-1) und (2,n-1) geblasen:

100	100	100	100	100	100	100	100	100	100	100
100	100	100	100	100	100	100	100	100	100	100
100	100	100	100	100	100	100	100	100	100	100
100	100	100	100	100	100	100	100	100	100	100
100	100	100	100	100	100	100	100	100	100	100
100	100	100	100	100	100	100	100	100	100	100
100	100	100	100	100	100	100	100	100	100	100
100	100	100	100	100	100	100	100	100	100	100
100	100	140	134	131	131	112	110	100	100	100
100	111	600	0	0	0	0	0	0	0	100

Anschließend werden die Blätter von Feld (2, n-1) mit einem Muster auf das Nicht-Randfeld (2, n-2) transferiert, und die Ecken (0, n-1) sowie (n-1, n-1) werden geleert bzw. das Laub dieser Ecken wird auf die Felder (0, n-2) sowie (n-1, n-2) transferiert. Hierdurch erreicht man eine teilweise vollständige und teilweise asymptotische Leerung der unteren Reihe:

100	100	100	100	100	100	100	100	100	100	100
100	100	100	100	100	100	100	100	100	100	100
100	100	100	100	100	100	100	100	100	100	100
100	100	100	100	100	100	100	100	100	100	100
100	100	100	100	100	100	100	100	100	100	100
100	100	100	100	100	100	100	100	100	100	100
100	100	100	100	100	100	100	100	100	100	100
100	100	100	100	100	100	100	100	100	100	100
127	100	169	130	138	123	107	101	180	128	
0	0	2	0	0	0	0	0	0	2	

Dasselbe Verfahren wird nun angewendet, um die Ränder rechts und links zu leeren:

100	100	100	100	100	100	100	100	100	100	100
2	100	100	100	100	100	100	100	100	100	0
0	223	100	100	100	100	100	100	170	0	0
2	129	100	100	100	100	100	100	125	2	0
0	132	100	100	100	100	100	100	137	0	0
0	125	100	100	100	100	100	100	124	0	0
0	119	100	100	100	100	100	100	119	0	0
0	108	100	100	100	100	100	100	108	0	0
0	100	204	143	127	126	125	110	100	0	0
2	1	0	1	0	0	0	0	0	2	0

Abschließend wird das Laub der Nicht-Randfelder auf den Feldern, die sich in derselben Spalte wie Q befinden, konzentrieren, indem das Laub vom rechten und vom linken Rand aus dorthin geblasen wird.

100	100	104	114	100	128	114	109	100	100
1	0	0	2	765	2	0	0	0	2
0	0	0	2	390	2	0	0	0	0
2	0	0	2	390	2	0	0	0	1
0	0	0	2	385	2	0	0	0	0
0	0	0	1	339	2	0	0	0	0
0	0	0	2	386	2	0	0	0	0
0	0	0	31	705	19	0	0	0	0
0	100	0	2	719	2	0	131	100	0
0	0	0	34	0	13	0	0	0	2

2. Phase (Das gesamte Laub in die oberste Reihe blasen):

An dieser Phase wurde fast nichts abgeändert. Das Laub wird weiterhin Zeile für Zeile nach oben geblasen, allerdings wird das Laub, das sich in der Spalte von Q befindet, nicht über Feld Q hinausgeblasen. Dadurch sieht der Hof nach Durchführung der 2. Phase so aus:

134	181	394	751	259	718	383	197	160	139
0	0	0	0	1045	0	0	0	0	0
0	0	0	0	374	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	18	0	16	0	0	0	0

Die restlichen Phasen bleiben alle unverändert. Die wesentliche Änderung erfolgte in Phase 1, wo das Laub zunächst in der Spalte von Q gesammelt wird, bevor es Richtung Q geblasen wird. Hierdurch wird verhindert, dass beim Transferieren des Laubs vom oberen zum unteren Rand durch Seitenabtriebe Blätter am Rand des Hofs verloren gehen: Wenn die Blätter nicht am Rand, sondern

in der Mitte des Hofs geblasen werden, dann können dort logischerweise auch keine Blätter verloren gehen.

Laufzeit

Auf die Laufzeit hat die Abänderung des Verfahrens zur Implementierung dieses Sonderfalls keine direkte Auswirkung. Eine indirekte Auswirkung ist allerdings, dass man durch das veränderte Verfahren bei einem Hof gleicher Größe jetzt deutlich mehr Blasoperationen braucht, um den vom abgeänderten Verfahren generierten generalisierten Programmablaufplan durchzuführen und man daher wahrscheinlich länger braucht, um die Simulation durchzuführen.

Erweiterung 2: Mehrere Laubbläser

Diese Erweiterung basiert ebenfalls auf dem 2. Ansatz. Ich habe mir überlegt: Was, wenn der Hausmeister einen Kollegen bittet, ihm beim Laubblasen zu helfen? Wie groß wären die Auswirkungen, die dies auf den Laubblasprozess hätte? Treten Synergieeffekte zwischen den beiden Laubbläsern auf, die den Laubblasprozess erleichtern?

Im Folgenden wird erläutert, wie ein zweiter Laubbläser hinzugefügt werden kann, der gleichzeitig bzw. zusammen mit dem ersten Laubbläser bläst. Es wird dabei davon ausgegangen, dass die beiden Blasgeräte baugleich sind und für sie somit die gleichen Blasregeln gelten. Außerdem wird festgelegt, dass die beiden Hausmeister nicht auf demselben Feld stehen können (es wäre einfach zu unrealistisch, dass zwei je einen Laubbläser haltende Hausmeister auf dasselbe Feld passen).

Auch für die zweite Erweiterung muss zunächst der Algorithmus, der eine Blasoperation simuliert, angepasst werden. Da die Hausmeister beide gleichzeitig blasen, gilt es ab sofort als „eine Blasoperation“, wenn die beiden Hausmeister ihre Laubbläser zum selben Zeitpunkt je einmal betätigen.

Der Algorithmus nimmt daher ab sofort nicht mehr nur die Argumente feld0 und blow_direction. Er simuliert jetzt stattdessen zwei Laubbläser, die gleichzeitig blasen, und nimmt daher die Argumente feld0_b1 (Feld 0 des ersten Laubbläzers), blow_direction_b1 (Blasrichtung des ersten Laubbläzers), feld0_b2 (Feld 0 des zweiten Laubbläzers) und blow_direction_b2 (Blasrichtung des zweiten Laubbläzers).

Um eindeutig zu definieren, wie die beiden Laubblasgeräte sich beeinflussen, müssen die folgenden drei Fälle betrachtet werden. Basierend auf den Blasregeln kann genau bestimmt werden, welche Felder durch den ersten Laubbläser beeinflusst werden, wenn er sich auf feld0_b1 bläst und in die Richtung blow_direction_b1 bläst, und welche Felder durch zweiten Laubbläser beeinflusst werden, wenn er sich auf das Feld feld0_b2 stellt und in die Richtung blow_direction_b2 bläst.

Allgemeine Regel:

Grundidee: Wie aus den Blasregeln deutlich hervorgeht, übt der Laubbläser auf Feld A mit Abstand am meisten Kraft aus, da dieses Feld vollständig geleert wird. Es wird daher vereinfachend davon ausgegangen, dass der erste Laubbläser kein Laub auf das Feld A des zweiten Laubbläzers blasen kann, da der zweite Laubbläser dieses Laub sofort wieder wegblasen würde (und umgekehrt).

Hierfür wird bei der Simulation der Blaswirkung des ersten Laubbläzers eine unsichtbare Barriere um das Feld A des zweiten Laubbläzers errichtet, die wie die Umrandung des Hofs behandelt wird.

Analog dazu wird bei der Simulation der Blaswirkung des zweiten Laubbläzers eine unsichtbare Barriere um das Feld A des ersten Laubbläzers errichtet.

Diese Modellierung entspricht nicht ganz der Realität: Wenn Laubbläser 1 Laub auf das Feld A von Laubbläser 2 bläst und sich die beiden Laubbläzer dabei parallel gegenüberstehen, dann ergibt es Sinn, dass Feld A von Laubbläser 2 wie die Hofumrandung behandelt wird, da es wahrscheinlich zu ähnlichen Seitenabtrieben wie beim Blasen von Laub gegen die Hofumrandung kommen wird. Stehen sich die Laubbläzer jedoch orthogonal gegenüber, dann ist diese Modellierung allerdings nicht realitätsgerecht. Zur Vereinfachung wird dies hingenommen – der Sonderfall, in dem die nicht realitätsgerechte Modellierung auftritt, spielt bei der Entwicklung einer Strategie für zwei Laubbläzer ohnehin eine untergeordnete Rolle.

Bei der Simulation der Blaswirkung des ersten Laubbläzers wird außerdem überprüft, ob der zweite Laubbläser in die Richtung der ersten Laubbläzers bläst (dem ist der Fall, wenn die Bedingung $manhattan_distance(feld0_b2, feld0_b1) > manhattan_distance(feld0_b1, feld0_b2 + blow_direction_b2)$

erfüllt ist). Wenn der zweite Laubbläser in die Richtung des ersten Laubbläzers bläst, dann würde es wenig Sinn machen, dass Seitenabtriebe des ersten Laubbläzers auf dem Feld 0 des zweiten Laubbläzers landen, obwohl von diesem Feld ein starker Luftstrom in die Gegenrichtung ausgeht. Daher wird in dem Fall auch um das Feld 0 des zweiten Laubbläzers eine unsichtbare Barriere errichtet. Analog dazu wird bei der Simulation der Blaswirkung des zweiten Laubbläzers vorgegangen.

Sonderfall: Feld A des ersten Laubbläzers überschneidet sich mit Feld A des zweiten Laubbläzers

Da $feld0_b1 \neq feld0_b2$ gibt es genau 2 Situationen, in denen der Fall auftreten kann:

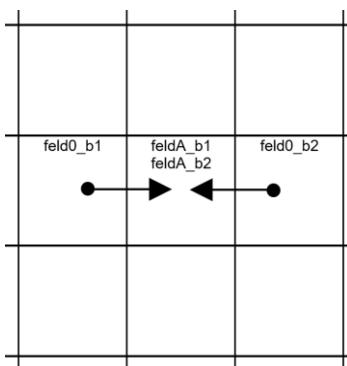


Abbildung 21: 1. Situation, in der $feldA_b1 = feldA_b2$ auftreten kann: Die Laubbläzer stehen sich gegenüber

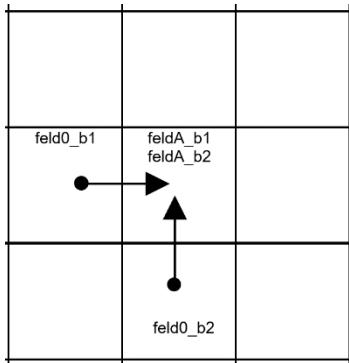


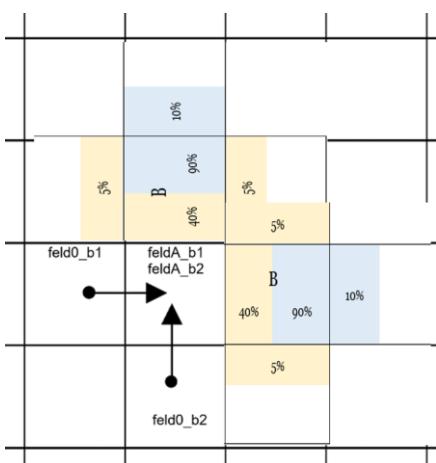
Abbildung 22: 2. Situation, in der $feldA_b1 = feldA_b2$ auftreten kann: Die Laubbläser stehen sich orthogonal gegenüber

Den allgemeinen Regeln zu folge würde in diesem Sonderfall nichts passieren: Für den ersten Laubbläser wäre Feld A blockiert, da es dem Feld A des 2. Laubbläasers entspricht – umgekehrt wäre für den 2. Laubbläser Feld A blockiert, da es dem Feld A des 1. Laubbläasers entspricht. Dies ist aber natürlich nicht realistisch.

Daher wird festgelegt, dass in diesem Sonderfall keine unsichtbare Barriere um Feld A errichtet wird. Stattdessen wird das sich auf dem gemeinsamen Feld A befindliche Laub zwischen den beiden Laubbläsern aufgeteilt. Jeder Laubbläser bläst dabei 50% des Laubs basierend auf der Blaswirkung seines Laubbläasers.

In der 1. Situation (Abb. 21) hat dies zur Folge, dass das gemeinsame Feld A vollständig geleert wird. Dadurch, dass Laubbläser 1 um $feld0_b2$ eine unsichtbare Barriere errichtet und Laubbläser 2 um $feld0_b1$ eine unsichtbare Barriere erreicht, wenden beide Bläser die Regeln für das Blasen auf Randfelder orthogonal zum Laub an, was dazu führt, dass 50% des Laubs auf das Feld über A und 50% des Laubs auf das Feld unter A gerät.

In der 2. Situation werden die ganz normalen Blasregeln aus der Aufgabenstellung für jeden Laubbläser mit jeweils 50% der Blätter von Feld A angewendet:



Synergien ausnutzen

Durch die **Möglichkeit, zwei Blasoperationen gleichzeitig durchzuführen**, können Phase 1 (Leeren der untersten Reihe) und Phase 2 (Transferieren des Laubs auf die oberste Reihe) deutlich schneller ausgeführt werden. Außerdem können **zwei Ecken parallel zueinander geleert werden**,

wenn jeder Hausmeister sich um eine Ecke kümmert. Nicht immer ist es sinnvoll, zwei Blasoperationen gleichzeitig auszuführen – z.B. kann es passieren, dass die Anzahl an zu leerenden Felder in Phase 2 nicht gerade ist und daher bei der letzten Blasoperation der Hausmeister nichts zu tun hat. Es gibt grundsätzlich diese Möglichkeiten, mit den Hausmeister, der nichts zu tun hat, umzugehen:

- Man könnte den Hausmeister eine Pause machen lassen,
- Man könnte den Hausmeister auch genauso gut irgendwo in die Wüste schicken und dort irgendeiner sinnlosen Beschäftigung zuführen.
- Sinnvoller ist es aber, Blasoperationen festzulegen, die immer entweder gar keinen Effekt haben oder zumindest eine geringe Menge Laub auf Q befördern und daher zu jedem Zeitpunkt ohne Probleme ausgeführt werden können. Sollte der Hausmeister mal nichts zu tun haben, dann kann er eine dieser Blasoperationen zufällig auswählen und ausführen

Deutlich schnelleres Aufräumen von Randfeldern möglich: Ein Randfeld kann nun deutlich schneller geleert werden, indem diese Blasoperation in einem Muster ausgeführt wird:

(0,0)	(1,0)	(2,0)	(3,0)
(0,1)	(1,1)	(2,1)	(3,1)
(0,2)	(1,2)	(2,2)	(3,2)

Abbildung 23: Eine Blasoperation ist eingezeichnet. Die beiden Punkte markieren `feld0_b1` und `feld0_b2`, die beiden Pfeile markieren `blow_direction_b1` und `blow_direction_b2`. Das orangene Randfeld wird geleert, Target-Feld ist das grüne Feld

Deutlich schnelleres Transferieren des Laubs von Feld (Q[0],1) auf (Q[0],2) möglich: Das Laub kann sehr effektiv transferiert werden, indem in einem Muster abwechselnd die rote und die schwarze Blasoperation durchgeführt werden (siehe Abbildung):

(Q[0]-1,0)	(Q[0],0)	(Q[0]+1,0)
(Q[0]-1,1)	(Q[0],1)	(Q[0]+1,1)
(Q[0]-1,2)	(Q[0],2)	(Q[0]+1,2)

Laufzeit

Die theoretische Laufzeit des Algorithmus verändert sich durch diese Erweiterung nicht. Die praktische Evaluierungsdauer verbessert sich jedoch, da mit zwei Laubblasern weniger Blasoperationen simuliert werden müssen, bis der gewünschte Zielzustand erreicht ist.

j Laubbläser

Eine Implementierung von j verschiedenen Laubbläsern (für beliebige natürliche j), die alle gleichzeitig blasen, habe ich nicht vorgenommen. Eine solche Erweiterung wäre aber im Grunde genommen nicht schwer umzusetzen; die einzige wesentliche Änderung bestünde darin, dass jeder Laubbläser vor der Simulation der Blaswirkung Barrieren um alle Felder A aller anderen Laubbläser errichten müsste (wie im Vorkapitel für zwei Laubbläser erläutert).

Trotzdem steht die theoretische Frage im Raum: **Welche zusätzlichen Synergieeffekte würden entstehen, wenn man 3 oder mehr Laubbläser gleichzeitig blasen ließe?**

- Wenn drei Laubbläser gleichzeitig blasen, dann entsteht hierdurch erstmals die Möglichkeit, das gesamte Laub von einem beliebigen Nicht-Randfeld in nur einem Schritt auf ein anderes Feld zu blasen. Hierfür müssen die Laubbläser U-förmig um das zu leerende Feld angeordnet werden, d.h. die Laubbläser stellen sich auf die drei Nachbarfelder vom zu leerenden Feld, auf die das Laub nicht gelangen soll, nur das Feld, auf das das Laub geblasen werden soll, wird offen gelassen. Das zu leerende Feld wird dabei vollständig geleert.
Dies hat zur Folge, dass jedes beliebige Nicht-Randfeld nun mit geringem Aufwand vollständig geleert werden kann. Hierdurch wird der letzte Teil von Phase 4 deutlich einfacher.
- Wenn noch mehr Laubbläser hinzukommen, dann beschleunigt das zwar natürlich den Blasprozess und verstärkt bestehende Synergien, es entstehen aber keine wesentlichen neuen Synergieeffekte mehr. Der „letzte große Sprung“ findet also beim Erhöhen der Laubbläserzahl von 2 auf 3 statt.
- Bei einem Hof mit fixen Seitenlängen ist irgendwann der Punkt erreicht, wo eine weitere Erhöhung der Laubbläserzahl keinen Sinn mehr macht, da die neuen Laubbläser nur alte blockieren würden (spätestens dann, wenn es mehr Bläser als Felder gibt, ist dieser Punkt auf jeden Fall erreicht).

Erweiterung 3: Verschiedene Laubtypen

Die Erweiterung 3 führt zwei verschiedene Laubtypen ein, die jeweils unterschiedlichen Regeln unterliegen. Dies erfordert Änderungen der Datenstrukturen, die die Blattverteilungen speichern, und der Funktionen, die die Blasoperationen durchführen.

Diese Erweiterung ist besonders interessant wegen ihrem Realitätsbezug: Auf einem Schulhof wird in der Regel nicht nur monokulturartig eine Baumart gepflanzt. Stattdessen muss davon ausgegangen werden, dass verschiedene Baumarten wachsen und somit verschiedene Laubtypen wegzublasen sind.

Die Erweiterung wird so umgesetzt, dass für jede Blattart die geltenden Blasregeln angewendet werden können. Bei der Simulation des Blasvorgangs wird anschließend auf jede Blattart die passende Regel angewendet.

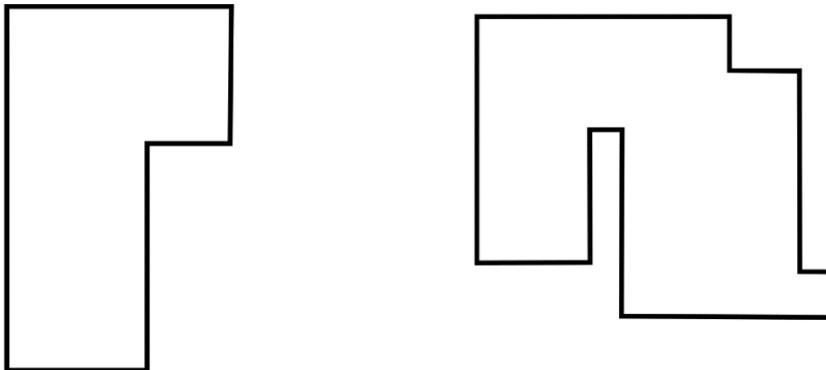
Weitere Erweiterungsideen (mit Lösungsansätzen)

Andere Hofformen: Rechtwinklige Polygone

Hierbei handelt es sich um eine Erweiterungsidee, die eigentlich implementieren wollte, aufgrund von Zeitdruck (Abiturvorbereitung) aber leider abbrechen musste. Ich werde hier trotzdem

beschreiben, wie die erweiterte Aufgabe ausgesehen hätte und was für eine Art von Lösung ich implementiert hätte.

Ziel war die Entwicklung eines Verfahrens, dass auch auf Höfen, die nicht rechteckig geformt sind, so viel Laub wie möglich auf Q sammeln kann. Die Höfe sollten die Form eines Polygons haben, das ausschließlich rechte Innen- oder Außenwinkel hat. Beispiele für solche Formen:



Ich plante, zur Lösung dieses Problems einen Divide-and-conquer Algorithmus zu verwenden. Dieser Algorithmus sollte das Polygon zunächst in zusammenhängende Rechtecke aufteilen, die in einer Baumstruktur (Graph) gespeichert werden, damit ersichtlich ist, welches Rechteck sich mit welchen Rechteck überlappt (Divide-Teil):

Algorithmus 12: Divide-Teil des Divide-and-conquer Algorithmus

Parameter: $Q = \text{Index von Feld } Q$

Parameter: $\text{Hof} = \text{Polygon, das den Hof darstellt}$

1. $\text{rechteck_tree} = \text{Initialisiere einen leeren Baum}$

2. $\text{initial_rechteck} = \text{Finde das größte Rechteck, das in das Polygon Hof passt und } Q \text{ beinhaltet}$

3. Füge initial_rechteck als erste bode zu rechteck_tree hinzu

4. **Endlosschleife, die bis zum Abbruch ausgeführt wird:**

5. $\text{unbound_field} = \text{Finde ein Feld, dass Nachbarfeld eines der sich in } \text{rechteck_tree} \text{ befindenden Rechtecke ist und noch auf keinem der in Hof vorhandenen Rechtecke liegt}$

6. $\text{origin_rechteck} = \text{Finde das Rechteck, von dem } \text{unbound_field} \text{ ein Nachbarfeld ist}$

7. $\text{new_rechteck} = \text{Finde das größte Rechteck, das in das Polygon Hof passt, } \text{origin_rechteck} \text{ schneidet und } \text{unbound_field} \text{ beinhaltet}$

8. **füge new_rechteck am Baum rechteck_tree an der Node von origin_rechteck an**

...

Anschließend werden die im Baum gespeicherten Rechtecke „von außen nach innen“ geleert. Das bedeutet, es wird mit der optimalen Leerung des Rechtecks, das im Baum am weitesten vom Rechteck initial_rechteck entfernt ist, begonnen. Das Laub dieses Rechtecks wird auf dem Feld gesammelt (unter Verwendung von Ansatz 2), das das Rechteck mit dem Rechteck, das im Baum durch die Parent-Node des zu leerenden Rechtecks repräsentiert ist, schneidet. Dieser Prozess wird so lange fortgesetzt (in jeder Iteration wird immer das noch nicht geleerte Rechteck ausgewählt, das im Baum am weitesten von initial_rechteck entfernt ist), bis alle Rechtecke aus dem Baum bis auf

das Rechteck, das Q enthält, leer sind. Dieses Rechteck kann nun unter Verwendung von Ansatz 2 so aufgeräumt werden, dass das Laub auf Q geblasen wird.

Weitere Ideen

Ich hatte noch viele weitere Ideen für mögliche Erweiterungen, die aus zeitlichen Gründen nicht umsetzbar waren:

- Höfe mit sechseckigen Feldern
- Seitenwinde
- Eine Variante der Aufgabe, bei der der Hausmeister den Laubbläser dauerhaft eingeschaltet lassen muss

Umsetzung

Die Lösungsidee wird in Python 3.10 implementiert. Verwendete Python-Bibliotheken:

- *math*: Verwendet für mathematische Operationen (atan, Quadratwurzel, comb), Python standard library
- *matplotlib*: Verwendet zum Visualisieren des Hofs in einem Plot
- *random*: Simulieren des Zufalls
- *numpy*: Modellierung des Hofs als numpy-Array

Insgesamt werden diese Dateien erstellt:

Ordner	Datei	Inhalt
Aufgabe 1	aufgabe1_solver1.py	Implementiert den 1. Ansatz
	aufgabe1_solver2.py	Implementiert den 2. Ansatz (bester Ansatz bzw. zu bewertendes Verfahren)
	aufgabe1_solver3.py	Implementiert den 3. Ansatz
	hof.py	Enthält die Klassen Rules und Hof (Modellierung eines Schulhofs und der Regeln, die auf ihm gelten)
	binomial_util.py	Enthält Hilfsfunktionen zum Berechnen von Binomialverteilungen
Aufgabe 1 E1	aufgabe1_solver2.py	Erste Erweiterung (keine Hofmauer), basiert auf 2. Ansatz
	hof.py	Enthält die Klassen Rules und Hof (Modellierung eines Schulhofs und der Regeln, die auf ihm gelten)
	binomial_util.py	Enthält Hilfsfunktionen zum Berechnen von Binomialverteilungen
Aufgabe 1 E2	aufgabe1_solver2.py	Zweite Erweiterung (zwei Laubbläser), basiert auf 2. Ansatz

	hof.py	Enthält die Klassen Rules und Hof (Modellierung eines Schulhofs und der Regeln, die auf ihm gelten)
	binomial_util.py	Enthält Hilfsfunktionen zum Berechnen von Binomialverteilungen
Aufgabe 1 E3	aufgabe1_solver2.py	Dritte Erweiterung (mehrere verschiedene Laubtypen), basiert auf 2. Ansatz
	hof.py	Enthält die Klassen Rules und Hof (Modellierung eines Schulhofs und der Regeln, die auf ihm gelten)
	binomial_util.py	Enthält Hilfsfunktionen zum Berechnen von Binomialverteilungen

Im Folgenden werden die besonders wichtigen und interessanten Teile der Implementierung erläutert.

Hilfsfunktionen zur Berechnung von Binomialverteilungen (binomial_util.py)

Ich definiere die Funktion `binomialpdf(*, n, p, k)`, die den Wert von $P(X=k)$ mit den Parametern n und p und einer binomialverteilte Größe X berechnet und als *float* zurückgibt. Hierfür wird zunächst der Binomialkoeffizient mithilfe von *math library* bestimmt:

```
bincomb = math.comb(n, k)
```

Anschließend wird über die logarithmische Darstellung $P(X=k)$ berechnet und zurückgegeben.

```
log_binom = math.log(bincomb) + k * math.log(p) + (n - k) * math.log(1 - p)
return math.exp(log_binom)
```

Es wird außerdem die Funktion `binomialdist(*, n, p, relevant_threshold=0.01)` definiert, die $P(x|X=k)$ für alle k im Bereich $[0;n]$ und den Parametern n und p , die der Funktion als Keyword-Argumente gegeben werden, für eine binomialverteilten Größe X berechnet. Die Funktion gibt eine Liste *dist* mit der Wahrscheinlichkeitsverteilung zurück, der Index in der Liste korrespondiert mit dem jeweiligen k -Wert, dessen Eintrittswahrscheinlichkeit am entsprechenden Index in der Liste beschrieben wird.

Da es für große n sehr aufwändig wäre, alle $P(X=k)$ zu berechnen, wird stattdessen ein anderes Vorgehen genutzt: Beim Berechnen von $P(X=k)$ wird mit dem k -Wert begonnen, für den $P(X=k)$ maximal wird. Dann wird sich das Monotoniekriterium der Binomialverteilung zunutze gemacht:

```
dist = np.zeros(n+1) # Mit Nullen gefüllte Liste erzeugen
for k in range(math.floor(n*p), -1, -1):
    pdf = binomialpdf(n=n, p=p, k=k)
    dist[k] = pdf
    if pdf < relevant_threshold:
```

```

        break
for k in range(math.ceil(n*p), n+1, 1):
    pdf = binomialpdf(n=n, p=p, k=k)
    dist[k] = pdf
    if pdf < relevant_threshold: # der restliche Bereich ist
        vernachlässigbar, da die Wahrscheinlichkeiten
        verschwinden gering werden
        break
...

```

Sobald sehr kleine Werte von $P(X=k)$ werden nicht berechnet, stattdessen werden diese Listenelemente „abgeflacht“ bzw. es wird statt dem korrekten $P(X=k)$ Wert ein konstanter Wert in der Verteilungsliste *dist* gespeichert, sodass die Liste aufsummiert immer noch 1 ergibt:

```

...
if len(dist[dist==0]) != 0:
    fill_rest = (1 - sum(dist)) / len(dist[dist==0])
    dist[dist==0] = fill_rest
return dist

```

Es wird außerdem eine *binomial_likeliest* ($*, n, p, rank=0, handle_ties="higher"$) -Funktion definiert. Diese Funktion bestimmt den wahrscheinlichsten Fall, der bei einer Binomialverteilung mit $n=n$ und $p=p$ auftritt. Wird das Keyword Argument *rank* = „random“ übergeben, dann simuliert die Funktion den tatsächlichen Zufall bzw. wählt basierend auf der mit *binomialdist* berechneten Binomialverteilung als Wahrscheinlichkeits- bzw. Gewichtefunktion ein k mithilfe von *random.choices* zufällig aus:

```

def binomial_likeliest(*, n, p, rank=0, handle_ties="higher"):
    if rank == "random":
        dist = binomialdist(n=n, p=p)
        k = random.choices(population=[i for i in range(n+1)], k=1,
                           weights=dist)[0]
        return k, dist[k]

```

Definieren einer Rules-Klasse (hof.py)

Die Rules-Klasse dient zum Speichern der Regeln, die auf einem Hof gelten. Der Konstruktor der Klasse nimmt diese Keyword-Argumente, die als Attribute der Klasse gespeichert werden:

- *A_seitenabtrieb* = 0.1,
- *B_vorne_abtrieb* = 0.1,
- *A_noB_seitenabtrieb* = $0.5 * 0.95$,
- *s1* = 0.9,
- *s4* = 0.05,
- *use_binomial* = True,
- *binomial_rank* = "random",
- *binomial_handle_ties* = "higher"

Noch im Konstruktor der Klasse wird sichergestellt, dass die angegebenen Argumente alle im gültigen Wertebereich liegen:

```
...
    assert 0 < A_seitenabtrieb < 0.5 # Implementierungsbedingte
Einschränkung: An dieser Stelle wird festgelegt, dass niemals alle Blätter auf
abgetrieben werden
    assert 0 < B_vorne_abtrieb < 1
    assert 0 < A_noB_seitenabtrieb <= 0.5 # Implementierungsbedingt wird in
diesem Fall allerdings erlaubt, dass das ganze Laub zur Seite wegfliegt
    ...
    assert isinstance(binomial_rank, int) or binomial_rank == "random"
    assert binomial_handle_ties == "higher" or binomial_handle_ties ==
"lower" or binomial_handle_ties == "random"
```

Definieren einer Hof-Klasse (hof.py)

Es wird außerdem eine Klasse namens Hof erstellt, die einen aus Planquadraten bestehenden Hof repräsentiert, auf dem Laub geblasen werden kann.

Attribute der Klasse:

Hof.x_size – speichert die x-Größe des Hofs

Hof.y_size – speichert die y-Größe des Hofs

Hof.rules – speichert die ein Rules-Objekt, in dem die Regeln festgesetzt sind, die für Blasoperationen auf dem Hof gelten

Hof.startwert – speichert die Anzahl an Blättern, die sich zu Beginn auf jedem Feld befindet

Hof.blas_counter – Integer, der die Anzahl an auf dem Hof durchgeföhrten Blasoperationen zählt

Hof.blas_log – Liste, in der die durchgeföhrten Blasoperationen als dictionaries gespeichert werden

Funktionen der Klasse:

Hof.render(, title="“, plot_last_op=False)* – plottet den Hof mit matplotlib und zeigt ihn in einem Popup an. Dabei wird Feld Q schwarz umkastet. Felder, die vollständig geleert sind, werden orange eingefärbt. Felder, die nicht vollständig eingefärbt sind, werden grün mit verschiedenen Sättigungsstufen eingefärbt. Dabei entspricht die Sättigungsstufe der Blattmenge auf dem Feld geteilt durch die Blattmenge auf dem Feld, auf dem am meisten Blätter liegen:

```
saturation = min(1, self.felder[i][j] / max_value)
```

Über das *title* Argument kann die Überschrift des Plots eingestellt werden. Wenn die zuletzt durchgeföhrte Blasoperation als Pfeil geplottet werden soll, dann muss *plot_last_op* auf True gesetzt werden.

Hof.__str__(, round_digits=5)* – gibt eine formatierte Darstellung des Hofs zurück, der entnommen werden kann, wie viele Blätter sich auf den jedem Feld befinden.

Hof.print_blas_log() – Gibt den Blas-Log als String zurück

Hof.is_corner(feld : tuple) – Gibt als boolean zurück, ob das Feld am Index feld ein Eckfeld ist

Hof.is_edge(feld : tuple) – Gibt als boolean zurück, ob das Feld am Index feld ein Randfeld ist

Hof.does_exist(feld : tuple) – Gibt als boolean zurück, ob das Feld am Index feld im Hof existiert

Hof.are_adjacent(feldA: tuple, feldB : tuple) – Gibt als boolean zurück, ob feldA Nachbarfeld von feldB ist

Hof.orthogonal_direction(direction) – Gibt mit folgendem Code einen Richtungsvektor zurück, der zu direction orthogonal ist:

```
return (0,1) if direction[1] == 0 else (1,0)
```

Es wird außerdem eine *blas(feld0, blow_direction)-Funktion* implementiert, die basierend auf den Blasregeln einen Blasvorgang simuliert. Die Funktion loggt die Blasoperation, danach ermittelt sie zunächst die Richtung, die orthogonal zur Blasrichtung ist, und anschließend Feld A und Feld B:

```
# Richtung, die orthogonal zur Blasrichtung ist, ermitteln:
orthogonal_direction = self.get_orthogonal_direction(blow_direction)

# Feld A (Feld unmittelbar vor dem Laubbläser) ermitteln:
feldA = (feld0[0]+blow_direction[0], feld0[1]+blow_direction[1])

if not self.does_exist(feldA):
    return # -> Es gibt kein Feld vor dem Laubbläser bzw. der Laubbläser bläst gegen die
          # Umrandung. Für diesen Fall ist definiert, dass sich die Verteilung des
          # Laubs nicht verändert
new_feldA_value = 0 # In dieser Variable wird die neue Anzahl an Blättern auf Feld A
                     # gespeichert

# Feld B (Feld hinter Feld A) ermitteln:
feldB = (feldA[0]+blow_direction[0], feldA[1]+blow_direction[1])
```

Anschließend überprüft sie, ob Feld B existiert:

```
if self.does_exist(feldB):
```

Je nachdem ob B existiert werden basierend auf den entsprechenden Regeln die Seitenabtriebe bestimmt, danach werden alle betroffenen Felder aktualisiert (sofern sie existieren).

Bei der Bestimmung der Blattanzahl, die abgetrieben wird, wird dabei – je nachdem was in Hof.rules festgelegt ist – der Erwartungswert oder eine Zufallssimulation über die Binomialverteilung verwendet, was z.B. hier sichtbar ist:

```
if self.rules.use_binomial:
    A_noB_seitenabtrieb_1, _ = binomial_likeliest(n=self.felder[feldA],
                                                p=self.rules.A_noB_seitenabtrieb,
                                                rank=self.rules.binomial_rank,
                                                handle_ties=self.rules.binomial_handle_ties)
    A_noB_seitenabtrieb_2, _ = binomial_likeliest(n=self.felder[feldA] -
                                                A_noB_seitenabtrieb_1, p=self.rules.A_noB_seitenabtrieb/(1-
                                                self.rules.A_noB_seitenabtrieb), rank=self.rules.binomial_rank,
                                                handle_ties=self.rules.binomial_handle_ties)
else:
    A_noB_seitenabtrieb_1 = self.felder[feldA] *
                            self.rules.A_noB_seitenabtrieb
    A_noB_seitenabtrieb_2 = A_noB_seitenabtrieb_1
```

Da der dieser Funktion zugrundeliegende Algorithmus in der Lösungsidee bereits sehr ausführlich als Pseudocode analysiert wurde, wird auf eine weitere Analyse der Funktion verzichtet.

Grundaufgabe Ansatz 1

Es handelt sich hierbei nicht um meinen besten bzw. den zu bewertenden Ansatz, daher bleibt die Analyse relativ oberflächlich.

Die Parameter werden in Zeile 24 bis 36 im Code festgelegt:

```
# Hof-Eigenschaften festlegen:
Q = (3,4) # Index von Feld Q festlegen
hof_size = (7,7) # Hofseitenlängen festlegen
startwert = 100 # Anfangsanzahl an Blättern pro Feld

# Wahrscheinlichkeitsmodellierung festlegen:
use_binomial = True # Festlegen, ob die Wahrscheinlichkeiten basierend auf der
Binomialverteilung simuliert oder ob die Erwartungswerte verwendet werden sollen

weight_avg = 0.1 # Gewichtung des 1. Heuristik-Maßes (durchschnittlicher Laubabstand zu
Feld Q)
weight_varianz = 0.9 # Gewichtung des 2. Heuristik-Maßes (Varianz der Laubabstände zu
Feld Q)

# Abbruchbedingungen festlegen:
satisfied_constraint = 0.8 # Bei erreichen dieser prozentualen Laubmenge (im Verhältnis
zum Gesamtblaub) wird das Programm auf jeden Fall abgebrochen
max_operations = 300 # Maximal durchgeführte Anzahl an Operationen, nach denen der
Blasprozess auf jeden Fall abgebrochen wird

# Datei zum Speichern der durchgeföhrten Blasvorgänge festlegen:
output_file = ""
```

Auf oberster Ebene befinden sich zwei Hilfsfunktionen (*squared_std* und *manhattan_distance*). Es wird außerdem eine Solver1-Klasse definiert, die alle zum Lösen des Hofs über die Greedy-Heuristik relevanten Funktionen enthält. Die *Solver1.edge_distance* Funktion gibt die Randdistanz bzw. die kleinste Verbindungsstrecke zwischen den Randfeldern feld0 und feld1, die nur über Rand- und Eckfelder läuft, zurück.

Die Greedy-Heuristik zum Leeren der Randfelder wird als Funktion namens *Solver1.greedy_edges* implementiert, die Greedy-Heuristik, die auf Nicht-Randfelder anzuwenden ist, wird als *Solver1.greedy_mid* definiert und nimmt als Argumente die Gewichte *weight_varianz* und *weight_avg*.

Die *Solver1.step()* Funktion führt bei Aufruf den nächsten Schritt auf, der basierend auf einer der beiden Greedy-Heuristiken ausgewählt wird. Dabei speichert das *Solver1.clear_edges* Attribut, ob als nächstes ein Randfeld oder ein Nicht-Randfeld zu bearbeiten ist:

```
def step(self):
    if self.clear_edges:
        next_op = self.greedy_edges()
    else:
        next_op = self.greedy_mid(weight_varianz=self.weight_varianz,
weight_avg=self.weight_avg)
```

Wenn eine der Abbruchbedingungen erfüllt ist wird False, ansonsten wird True zurückgegeben:

```
self.hof.blase(next_op["feld0"], next_op["blow_direction"])
if self.satisfied_constraint is not None:
    if self.hof.felder[self.Q] / self.sum_laub >=
self.satisfied_constraint:
```

```

        return False
    if self.max_operations is not None:
        if self.hof.blas_counter >= self.max_operations:
            return False
    return True

```

Grundaufgabe Ansatz 2:

Auch hier werden zunächst in Zeile 8 bis 17 die Hyperparameter des Programms festgelegt:

```

# Hyperparameter festlegen
Q = (2,2) # Index von Feld Q festlegen
hof_size = (5,5) # Hofseitenlängen festlegen
use_binomial = True # Festlegen, ob die Wahrscheinlichkeiten basierend auf der Binomialverteilung
simuliert oder ob die Erwartungswerte verwendet werden sollen
tolerated_amount = 5 # Blattmenge, die auf nicht vollständig leerbaren Feldern als vernachlässigbar
gilt
max_muster_operations = 5000 # Anzahl an Operation, die pro Muster maximal durchgeführt
werden
startwert = 100 # Anfangszahl an Blättern pro Feld
choose_faster_path = True
# Datei zum Speichern der durchgeföhrten Blasvorgänge festlegen:
output_file = ""

```

Es wird eine *Muster* Klasse erstellt, in der das Muster-Konzept implementiert ist. Dabei wird im *Muster.operations* Attribut die Abfolge an Blasoperationen des Musters als Liste gespeichert. Eine einzelne Blasoperation wird hierbei als dictionary der Form {„feld0“: ..., „blow_direction“:...} gespeichert.

Das *Muster.next_op_index* Attribut speichert den Index, den die als nächstes auszuförende Operation in *Muster.operations* hat.

Die *Muster.reset()* Methode setzt das Muster auf seinen Anfangszustand zurück.

Die *Muster.step()* Methode führt basierend auf *Muster.operations* die nächste Operation des Musters aus, hierfür wird überprüft, ob diese Operation eine Blasoperation oder ein anderes Muster ist:

```

if isinstance(self.operations[self.next_op_index], dict):
    # -> Eine Blasoperation liegt vor, die ausgeführt wird
    self.hof.blase(self.operations[self.next_op_index]["feld0"],
self.operations[self.next_op_index]["blow_direction"])
    self.next_op_index += 1
    self.num_operations += 1
elif isinstance(self.operations[self.next_op_index], Muster):
    # -> Ein anderes Muster liegt vor, das ausgeführt wird
    run_another_step = self.operations[self.next_op_index].step()
    if run_another_step:
        return True
    self.next_op_index += 1
    self.num_operations += 1

if self.next_op_index == len(self.operations):
    # -> Einmal durch alle Operationen des Musters durchgelaufen -> i
zurücksetzen
    self.next_op_index = 0
    if self.check_for_changes:
        # Überprüfen, ob noch Veränderungen stattfinden

```

```

new_sum = sum([self.hof.felder[index] for index in self.source_fields])
if new_sum == self.current_sum:
    return False
self.current_sum = int(new_sum)
if isinstance(self.operations[self.next_op_index], Muster):
    self.operations[self.next_op_index].reset()

```

Anschließend werden die Abbruchbedingungen überprüft:

```

if self.num_operations >= self.num_max_operations:
    return False

if max([self.hof.felder[index] for index in self.source_fields]) <=
    self.tolerated_amount:
    return False

```

Die Solver2-Klasse speichert alle Informationen und Funktionen, die zum Erstellen und Ausführen eines generalisierten Ablaufplans benötigt werden. Der generalisierte Ablaufplan wird dabei in Solver2.strategy gespeichert.

Die folgenden Funktionen sind an der Erstellung des generalisierten Ablaufplans besonders stark beteiligt:

Solver2.add_operation(operation) – fügt eine Blasoperation oder ein Muster zum Ablaufplan hinzu und macht davor die Initialrotation rückgängig

Solver2.corner_to_edge(source_corner_field, target_field) – Gibt ein Muster zurück, das bei Anwendung Laub vom Eckfeld source_corner_field auf das Randfeld target_field bläst

Solver2.edge_to_mid (source_edge_field, target_field) – Gibt Muster zurück, das bei Anwendung Laub vom Randfeld source_edge_field und seinen beiden auf dem Rand liegenden Nachbarn auf das Nicht-Rand-oder-Eckfeld target_field bläst

Solver2.clear_bottom_line() – leert die unterste Zeile des Hofs (Phase 1)

Solver2.move_to_top_line() – leert die unterste Zeile des Hofs (Phase 2)

Solver2.concentrate_top_line() – leert die unterste Zeile des Hofs (Phase 3)

Solver2.transfer_to_Q() – leert die unterste Zeile des Hofs (Phase 4)

Solver2.build_strategy() – führt die Initialrotation durch und führt danach die vier zuvor genannten Funktionen aus

Beispiele – Grundaufgabe

Zum Demonstrieren der Funktionstüchtigkeit meiner Programme habe ich sie mit verschiedenen Parametern ausgeführt. Die vollständigen Beispielausgaben sowie die vom Programm erstellten Dateien, die die durchgeführten Blasoperationen enthalten, sind im Ordner *Aufgabe 1* zu finden.

Im Ordner Aufgabe 1/Outputs sind für jedes Beispiel die Blasoperationen zu finden, die die Programme bei der Simulation durchgeführt haben. Sie sind in Textdateien gespeichert. Dabei sind die Blasoperationen, die von Ansatz 1 bei Beispiel 1 durchgeführt wurden in der Textdatei „output_{ansatz}_{beispiel}.txt“ gespeichert.

Anmerkung: Wenn *use_binomial* True ist und *binomial_rank == „random“*, dann wird bei der Simulation der Blasvorgänge der tatsächliche Zufall simuliert (wie im Kapitel *Lösungsseite*

erläutert). Die Programmausgabe hängt somit vom Zufall ab und ist somit bei jedem Ausführen anders bzw. nicht reproduzierbar, auch wenn beim erneuten Ausführen dieselben Parameter gegeben werden.

Ansatz 1 (aufgabe1_solver.py)

Beispiel 1

Eingabeparameter:

```
Q = (2,2)
hof_size = (5,5)
use_binomial = True
binomial_rank = "random"
startwert = 100
weight_avg = 0.5
weight_varianz = 0.5
max_operations = 1000
satisfied_constraint = 0.8
```

Ausgabe in der Konsole:

AUSGANGSPUNKT:

Verwendete Wahrscheinlichkeitsmodellierung: binomial

Laubverteilung vor dem Blasprozess:

100		100		100		100		100
100		100		100		100		100
100		100		100		100		100
100		100		100		100		100
100		100		100		100		100

Führe Simulation durch ...

ERGEBNIS DER SIMULATION:

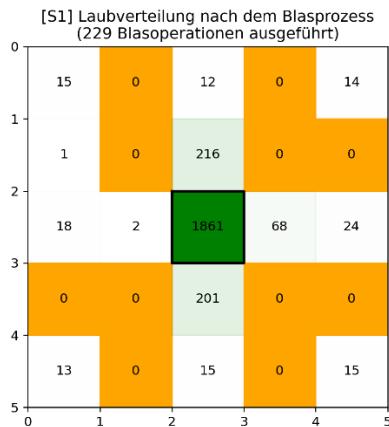
Ausgeführte Blasoperationen: 229

Anteil der Blätter auf Q an der Gesamtbläumenge: 74.44 %

Laubverteilung nach dem Blasprozess:

15		0		12		0		14
1		0		216		0		0
18		2		1861		68		24
0		0		201		0		0
13		0		15		0		15

Ausgabeplots:

**Beobachtung:**

Wie man sieht, gelingt es nicht, mehr als 74 % des Laubs auf Feld Q zu versammeln. Ansatz 2 schneidet da deutlich besser ab.

Beispiel 2**Eingabeparameter:**

$Q = (2,2)$

hof_size = (5,5)

use_binomial = True

binomial_rank = "random"

startwert = 100

weight_avg = 0.5

weight_varianz = 0.5

max_operations = 1000

satisfied_constraint = 0.8

Diese Eingabeparameter entsprechen denen aus Beispiel 1: Um zu zeigen, dass das Programm tatsächlich den Zufall simuliert, wurde es mit denselben Eingabeparametern zweifach ausgeführt.

Ausgabe in der Konsole:**AUSGANGSPUNKT:**

Verwendete Wahrscheinlichkeitsmodellierung: binomial

Laubverteilung vor dem Blasprozess:

100		100		100		100		100
100		100		100		100		100
100		100		100		100		100
100		100		100		100		100
100		100		100		100		100

Führe Simulation durch ...

ERGEBNIS DER SIMULATION:

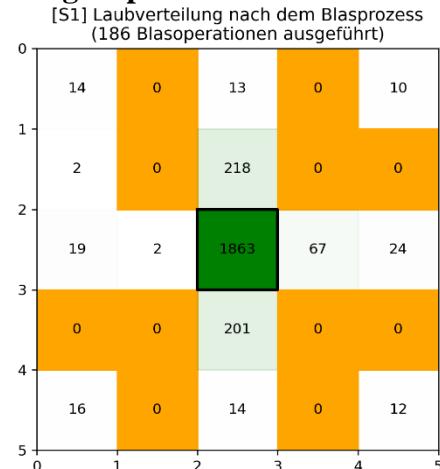
Ausgeföhrte Blasoperationen: 186

Anteil der Blätter auf Q an der Gesamtlaubmenge: 74.52 %

Laubverteilung nach dem Blasprozess:

14		0		13		0		10
2		0		218		0		0
19		2		1863		67		24
0		0		201		0		0

16		0		14		0		12
----	--	---	--	----	--	---	--	----

AusgabepLOTS:

Beispiel 3

Eingabeparameter:

$Q = (1,1)$
 hof_size = (3,4)
 use_binomial = True
 binomial_rank = "random"
 startwert = 100
 weight_avg = 0.5
 weight_varianz = 0.5
 max_operations = 1000
 satisfied_constraint = 0.8

Ausgabe in der Konsole:**AUSGANGSPUNKT:**

Verwendete Wahrscheinlichkeitsmodellierung: binomial

Laubverteilung vor dem Blasprozess:

100		100		100
100		100		100
100		100		100
100		100		100

Führe Simulation durch ...

ERGEBNIS DER SIMULATION:

Ausgeföhrte Blasoperationen: 455

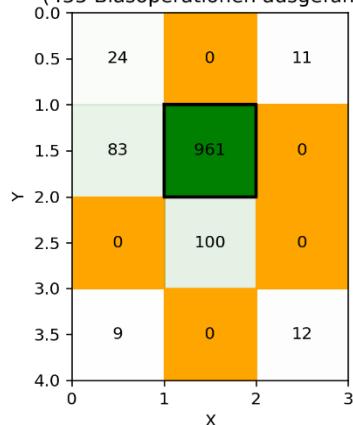
Anteil der Blätter auf Q an der Gesamtbläubmenge: 80.0833333333333 %

Laubverteilung nach dem Blasprozess:

24		0		11
83		961		0
0		100		0
9		0		12

AusgabepLOTS:

[S1] Laubverteilung nach dem Blasprozess
(455 Blasoperationen ausgeführt)



Beispiel 4

Eingabeparameter:

```
Q = (3,3)
hof_size = (8,8)
use_binomial = True
binomial_rank = "random"
startwert = 100
weight_avg = 0.5
weight_varianz = 0.5
max_operations = 1000
satisfied_constraint = 0.8
```

Ausgabe in der Konsole:

AUSGANGSPUNKT:

Verwendete Wahrscheinlichkeitsmodellierung: binomial

Laubverteilung vor dem Blasprozess:

100		100		100		100		100		100
100		100		100		100		100		100
100		100		100		100		100		100
100		100		100		100		100		100
100		100		100		100		100		100
100		100		100		100		100		100
100		100		100		100		100		100
100		100		100		100		100		100
100		100		100		100		100		100
100		100		100		100		100		100

Führe Simulation durch ...

ERGEBNIS DER SIMULATION:

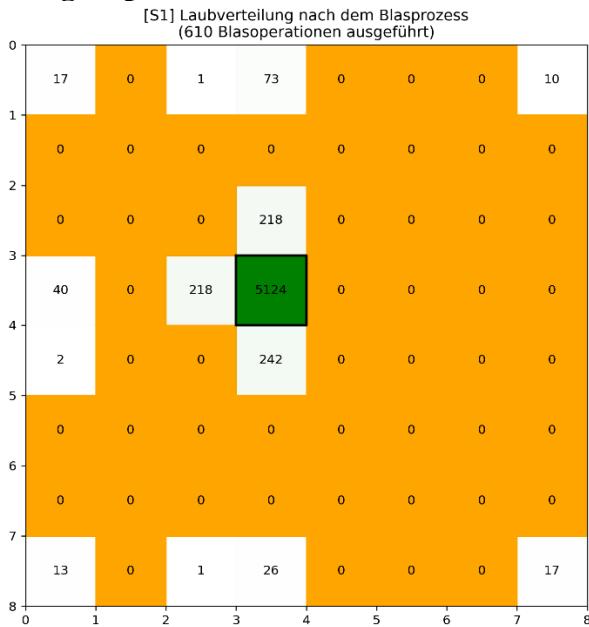
Ausgeführte Blasoperationen: 610

Anteil der Blätter auf Q an der Gesamtlaubmenge: 80.0625 %

Laubverteilung nach dem Blasprozess:

17		0		1		73		0		0
0		0		0		0		0		0
0		0		0		218		0		0
40		0		218		5124		0		0
2		0		0		242		0		0
0		0		0		0		0		0

0		0		0		0		0		0		0
13		0		1		26		0		0		17

Ausgabeplots:**Beobachtung:**

Auffällig ist, dass bereits bei einer Hofgröße von gerade einmal (8,8) das Programm bereits ca. 10 Sekunden braucht, bis die Simulation abgeschlossen ist. Dies ist auf die schlechte Laufzeit zurückzuführen. Bei den anderen Ansätzen, insbesondere bei Ansatz 2, ist die Laufzeit deutlich besser.

Beispiel 5**Eingabeparameter:**

```
Q = (3,4)
hof_size = (7,7)
use_binomial = True
binomial_rank = "random"
startwert = 100
weight_avg = 1
weight_varianz = 0
max_operations = 200
satisfied_constraint = 0.8
```

Ausgabe in der Konsole:**AUSGANGSPUNKT:**

Verwendete Wahrscheinlichkeitsmodellierung: binomial

Laubverteilung vor dem Blasprozess:

100		100		100		100		100		100		100
100		100		100		100		100		100		100
100		100		100		100		100		100		100
100		100		100		100		100		100		100
100		100		100		100		100		100		100
100		100		100		100		100		100		100
100		100		100		100		100		100		100

100		100		100		100		100		100
-----	--	-----	--	-----	--	-----	--	-----	--	-----

Führe Simulation durch ...

ERGEBNIS DER SIMULATION:

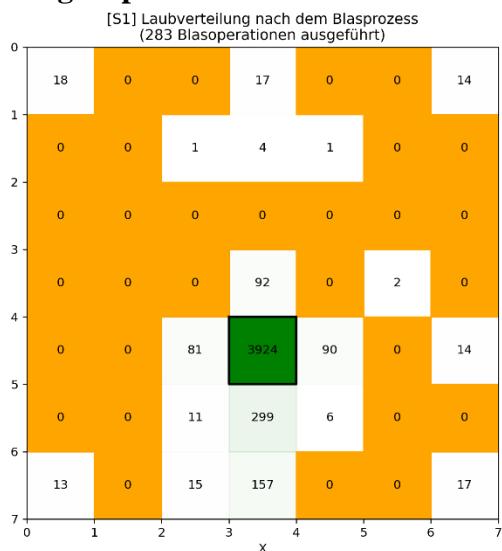
Ausgeführte Blasoperationen: 283

Anteil der Blätter auf Q an der Gesamtlaubmenge: 80.08163265306122 %

Laubverteilung nach dem Blasprozess:

18		0		0		17		0		0		14
0		0		1		4		1		0		0
0		0		0		0		0		0		0
0		0		0		92		0		2		0
0		0		81		3924		90		0		14
0		0		11		299		6		0		0
13		0		15		157		0		0		17

Ausgabeplots:



Beobachtung:

In diesem Beispiel wurden die Parameter so eingestellt, das maximal 300 Operationen durchgeführt werden. Das Gewicht für die Distanzheuristik wurde auf 1 und das für die Varianzheuristik auf 0 gestellt. Das Ergebnis ist, das in nur 300 Schritten relativ viel Laub auf Q versammelt wurde, das restliche Laub aber ungeordnet auf den restlichen Feldern verstreut ist.

Beispiel 6

Eingabeparameter:

$Q = (3,4)$

`hof_size = (7,7)`

`use_binomial = True`

`binomial_rank = "random"`

`startwert = 100`

`weight_avg = 0.1`

`weight_varianz = 0.9`

`max_operations = 200`

`satisfied_constraint = 0.8`

Ausgabe in der Konsole:

AUSGANGSPUNKT:

Verwendete Wahrscheinlichkeitsmodellierung: binomial

Laubverteilung vor dem Blasprozess:

100	100	100	100	100	100	100
100	100	100	100	100	100	100
100	100	100	100	100	100	100
100	100	100	100	100	100	100
100	100	100	100	100	100	100
100	100	100	100	100	100	100
100	100	100	100	100	100	100
100	100	100	100	100	100	100

Führe Simulation durch ...

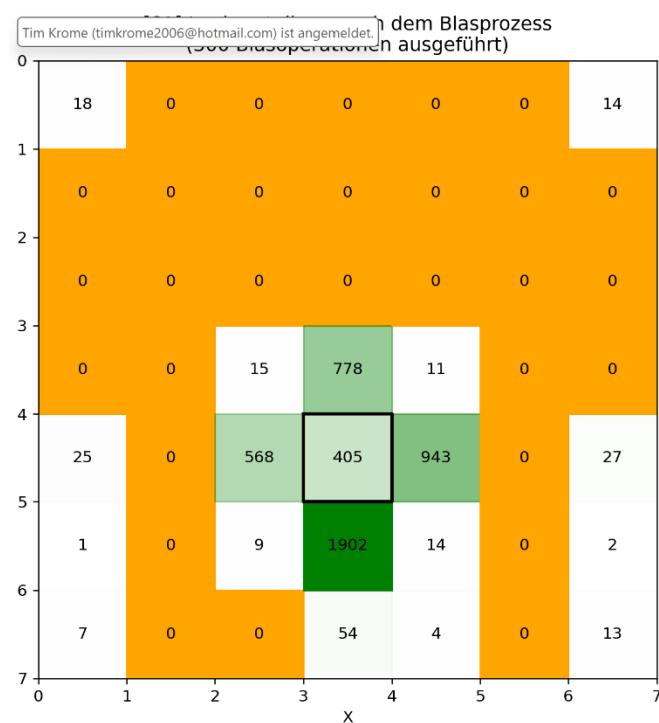
ERGEBNIS DER SIMULATION:

Ausgeföhrte Blasoperationen: 300

Anteil der Blätter auf Q an der Gesamtlaubmenge: 8.26530612244898 %

Laubverteilung nach dem Blasprozess:

18	0	0	0	0	0	0	14
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	15	778	11	0	0	0
25	0	568	405	943	0	0	27
1	0	9	1902	14	0	0	2
7	0	0	54	4	0	0	13

Ausgabeplots:**Beobachtung:**

In diesem Beispiel wurden die Parameter so eingestellt, das maximal 300 Operationen durchgeführt werden. Das Gewicht für die Distanzheuristik wurde auf 0.1 und das für die Varianzheuristik auf 0.9 gestellt. Das Ergebnis ist, das in nur 300 Schritten deutlich weniger Laub auf Feld Q gebracht wurde, das übrige Laub aber sehr geordnet um Feld Q herum angeordnet ist und bei Durchführung der entsprechenden Blasoperationen sehr schnell auf Feld Q gebracht werden könnte. (Da die Gewichtung der Distanzheuristik deutlich niedriger als die Gewichtung der Varianzheuristik ist, geschieht dies aber nicht).

Ansatz 2 (aufgabe2_solver.py)

Beispiel 1

Eingabeparameter:

```
Q = (2,2)
hof_size = (5,5)
use_binomial = True
tolerated_amount = 5
max_muster_operations = 5000
startwert = 100
choose_faster_path = True
```

Ausgabe in der Konsole:

AUSGANGSPUNKT:

Verwendete Wahrscheinlichkeitsmodellierung: binomial

Laubverteilung vor dem Blasprozess:

100		100		100		100		100
100		100		100		100		100
100		100		100		100		100
100		100		100		100		100
100		100		100		100		100

Führe Simulation durch ...

ERGEBNIS DER SIMULATION:

Ausgeföhrte Blasoperationen: 1686

Anteil der Blätter auf Q an der Gesamtbläubmenge: 98.24 %

Laubverteilung nach dem Blasprozess:

5		5		2		5		5
0		0		5		1		0
0		5		2456		5		0
0		0		0		0		0
2		0		0		0		4

GENERALISIERTER ABLAUFPLAN:

```
[Phase 1: Unterste Reihe entlauben]
blasen(Feld0: (4, 4), nach: links)
blasen(Feld0: (3, 4), nach: links)
blasen(Feld0: (2, 4), nach: links)
blasen(Feld0: (1, 4), nach: links)
Muster(Sourcefelder: [[(4, 4)]]) {
    blasen(Feld0: (3, 4), nach: rechts)
}
Muster(Sourcefelder: [[(0, 4)]]) {
    blasen(Feld0: (1, 4), nach: links)
}
```

[Phase 2: Laub auf oberste Reihe blasen]

```

blase(Feld0: (4, 4), nach: oben)
blase(Feld0: (3, 4), nach: oben)
blase(Feld0: (1, 4), nach: oben)
blase(Feld0: (0, 4), nach: oben)
blase(Feld0: (4, 3), nach: oben)
blase(Feld0: (3, 3), nach: oben)
blase(Feld0: (1, 3), nach: oben)
blase(Feld0: (0, 3), nach: oben)
blase(Feld0: (4, 2), nach: oben)
blase(Feld0: (3, 2), nach: oben)
blase(Feld0: (1, 2), nach: oben)
blase(Feld0: (0, 2), nach: oben)

[Phase 3: Laub auf oberster Reihe konzentrieren]
Muster(Sourcefelder: [[(0, 0)]]) {
    blase(Feld0: (0, 1), nach: oben)
}
Muster(Sourcefelder: [[(4, 0)]]) {
    blase(Feld0: (4, 1), nach: oben)
}

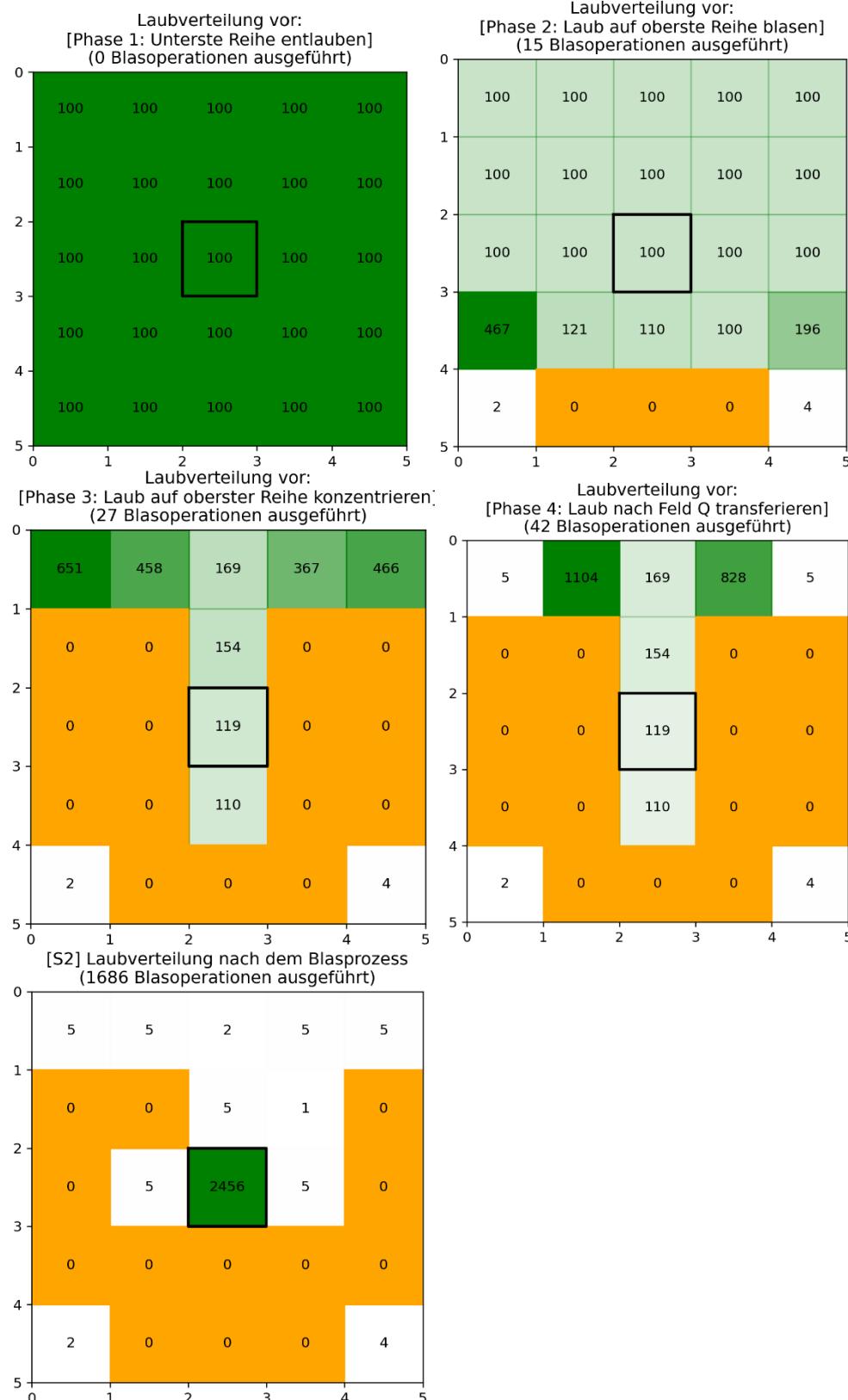
[Phase 4: Laub nach Feld Q transferieren]
Muster(Sourcefelder: [[(2, 0), (3, 0), (1, 0)]]) {
    blase(Feld0: (2, 1), nach: oben)
    blase(Feld0: (4, 0), nach: links)
    blase(Feld0: (0, 0), nach: rechts)
}
blase(Feld0: (2, 0), nach: unten)
blase(Feld0: (0, 2), nach: rechts)
blase(Feld0: (2, 4), nach: oben)
Muster(Sourcefelder: [[(1, 2)]]) {
    blase(Feld0: (1, 4), nach: oben)
}
Muster(Sourcefelder: [[(3, 2)]]) {
    blase(Feld0: (3, 4), nach: oben)
}
Muster(Sourcefelder: [[(1, 1), (2, 1), (3, 1), (2, 0)]]) {
    Muster(Sourcefelder: [[(1, 1), (2, 1), (3, 1)]]) {
        blase(Feld0: (0, 1), nach: rechts)
        blase(Feld0: (4, 1), nach: links)
    }
    Muster(Sourcefelder: [[(2, 0), (3, 0), (1, 0)]]) {
        blase(Feld0: (2, 1), nach: oben)
        blase(Feld0: (4, 0), nach: links)
        blase(Feld0: (0, 0), nach: rechts)
    }
}

```

AusgabepLOTS:

Aufgabe 1: Laubmaschen

Teilnahme-ID: 69408



Anmerkung: In späteren Beispielen wird teilweise nur noch der letzte Ausgabeplot, der den Zustand des Hofs nach Abschluss des Blasprozesses darstellt, abgebildet. Außerdem wird der generalisierte Programmablaufplan aus der Prorammausgabe herausgekürzt.

Beobachtung:

Wie man sieht, gelingt es dem Programm, fast das gesamte Laub auf Feld Q zu versammeln. Es braucht hierfür sehr viele Schritte, die fast alle in der letzten Phase ausgeführt werden. Dies ist dadurch begründet, dass ein Hof der Größe (5,5) mit $Q = (2,2)$ ein Sonderfall ist, der wie zuvor erläutert besonders kompliziert zu lösen ist.

Beispiel 2

Eingabeparameter:

```
Q = (1,3)
hof_size = (5,5)
use_binomial = True
tolerated_amount = 5
max_muster_operations = 5000
startwert = 100
choose_faster_path = True
```

Ausgabe in der Konsole:

AUSGANGSPUNKT:

Verwendete Wahrscheinlichkeitsmodellierung: binomial

Laubverteilung vor dem Blasprozess:

100	100	100	100	100
100	100	100	100	100
100	100	100	100	100
100	100	100	100	100
100	100	100	100	100

Führe Simulation durch ...

ERGEBNIS DER SIMULATION:

Ausgeführte Blasoperationen: 193

Anteil der Blätter auf Q an der Gesamtlaubmenge: 99.24 %

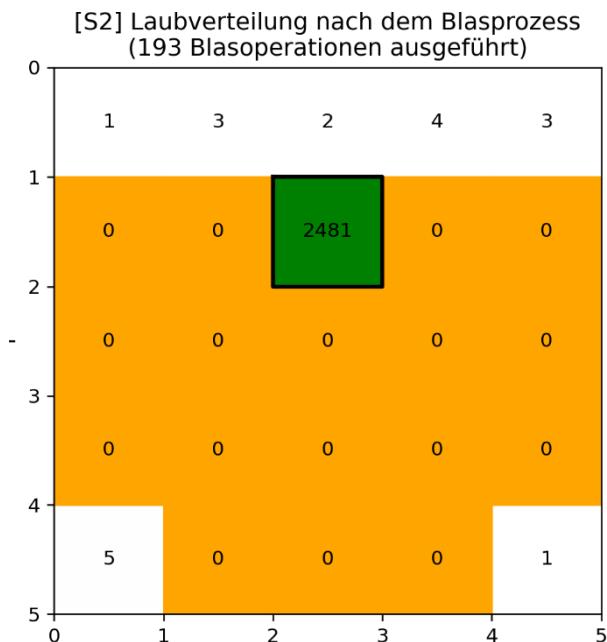
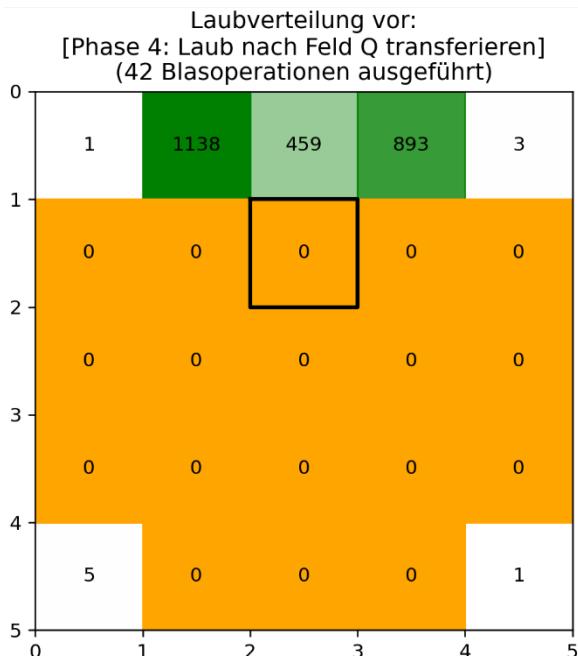
Laubverteilung nach dem Blasprozess:

1	3	2	4	3
0	0	2481	0	0
0	0	0	0	0
0	0	0	0	0
5	0	0	0	1

GENERALISIERTER ABLAUFPLAN:

(...)

Ausgabeplots:

**Beobachtung:**

Wie man sieht, nimmt Phase 4 deutlich weniger Blasoperationen weniger in Anspruch, wenn Feld Q nicht mehr das Feld in der Mitte, sondern ein Randfeld ist.

Beispiel 3**Eingabeparameter:**

$Q = (2,2)$
 $hof_size = (6,6)$
 $use_binomial = True$
 $tolerated_amount = 5$
 $max_muster_operations = 5000$
 $startwert = 100$
 $choose_faster_path = True$

Ausgabe in der Konsole:**AUSGANGSPUNKT:**

Verwendete Wahrscheinlichkeitsmodellierung: binomial

Laubverteilung vor dem Blasprozess:

100		100		100		100		100
100		100		100		100		100
100		100		100		100		100
100		100		100		100		100
100		100		100		100		100
100		100		100		100		100

Führe Simulation durch ...

ERGEBNIS DER SIMULATION:

Ausgeföhrte Blasoperationen: 402

Anteil der Blätter auf Q an der Gesamtbläubmenge: 98.80555555555556 %

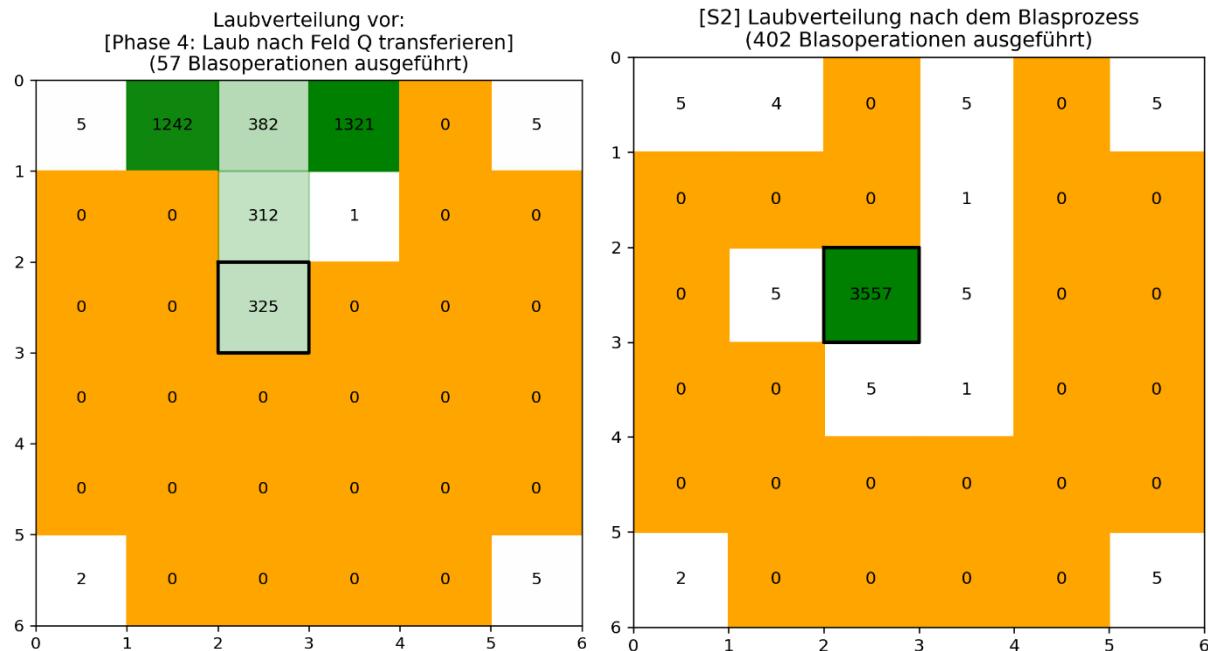
Laubverteilung nach dem Blasprozess:

5		4		0		5		0		5
---	--	---	--	---	--	---	--	---	--	---

0	0	0	1	0	0
0	5	3557	5	0	0
0	0	5	1	0	0
0	0	0	0	0	0
2	0	0	0	0	5

GENERALISIERTER ABLAUFPLAN:

(...)

Ausgabeplots:**Beobachtung:**

Im Vergleich zu Beispiel 1, wo ein Hof der Größe (5,5) aufgeräumt wurde, nimmt der Blasprozess bei einem Hof der Größe (6,6) deutlich weniger Blasoperationen in Anspruch. Dies ist ebenfalls darauf zurückzuführen, dass der Hof (5,5) mit $Q = (2,2)$ ein besonders schwerer Sonderfall ist.

Beispiel 4**Eingabeparameter:** $Q = (2,2)$ `hof_size = (6,6)``use_binomial = True``tolerated_amount = 5``max_muster_operations = 5000``startwert = 5000``choose_faster_path = True`**Ausgabe in der Konsole:****AUSGANGSPUNKT:**

Verwendete Wahrscheinlichkeitsmodellierung: binomial

Laubverteilung vor dem Blasprozess:

5000	5000	5000	5000	5000	5000
5000	5000	5000	5000	5000	5000
5000	5000	5000	5000	5000	5000
5000	5000	5000	5000	5000	5000
5000	5000	5000	5000	5000	5000
5000	5000	5000	5000	5000	5000

Führe Simulation durch ...

ERGEBNIS DER SIMULATION:

Ausgeföhrte Blasoperationen: 621

Anteil der Blätter auf Q an der Gesamtlaubmenge: 99.97722222222222 %

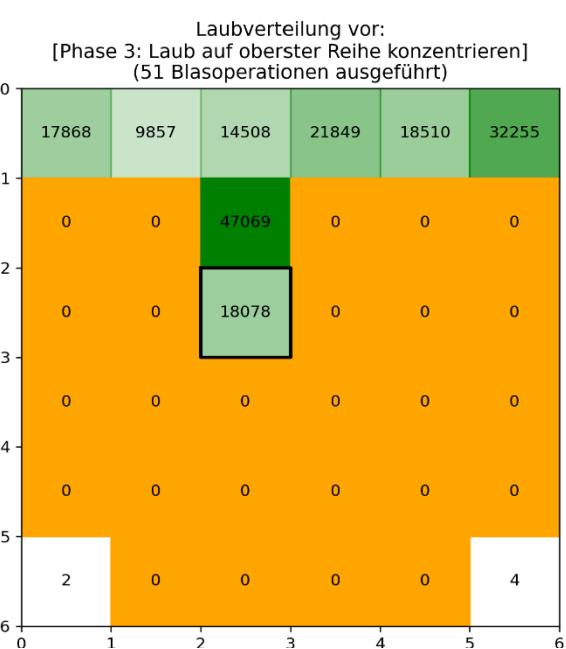
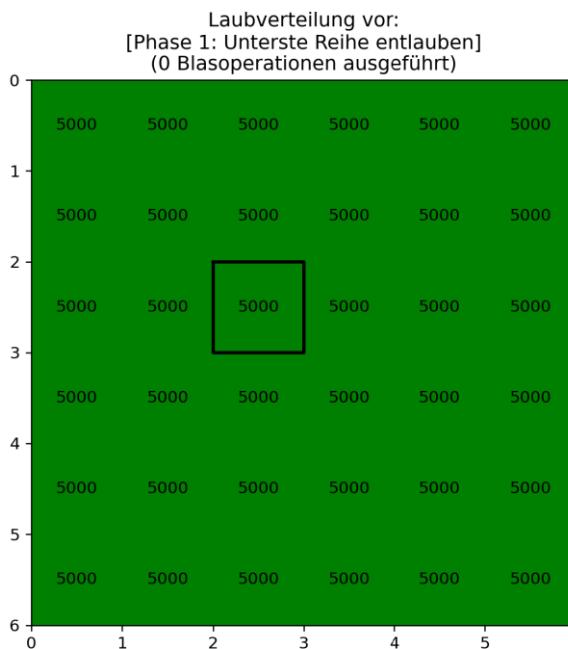
Laubverteilung nach dem Blasprozess:

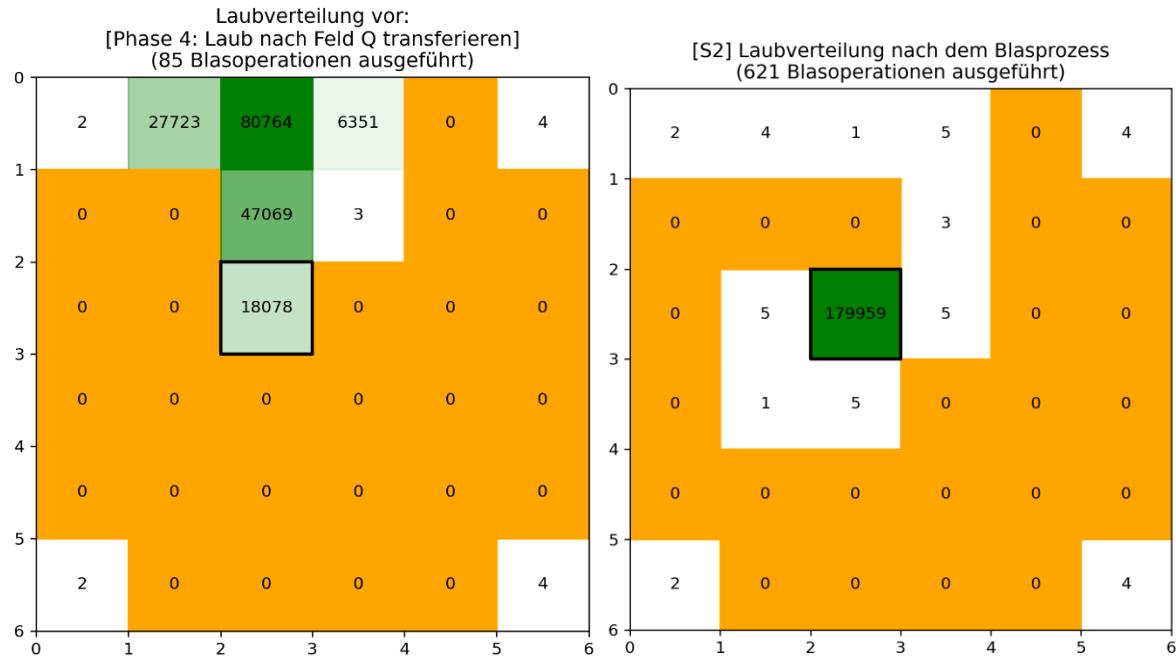
2	4	1	5	0	4
0	0	0	3	0	0
0	5	179959	5	0	0
0	1	5	0	0	0
0	0	0	0	0	0
2	0	0	0	0	4

GENERALISIERTER ABLAUFPLAN:

(...)

Ausgabeplots:



**Beobachtung:**

Auch bei Höfen, auf denen sich zu Beginn deutlich mehr Blätter auf den einzelnen Feldern befinden (in diesem Beispiel sind es zu Beginn 5 000 Blätter pro Feld), wird dank der logarithmischen Darstellung der Formel von Bernoulli in binomial_util.py innerhalb weniger Sekunden die Simulation durchgeführt. Im Vergleich zu Beispiel 3, wo sich auf jedem Feld zu Beginn 100 Blätter befanden, braucht das Programm hier ca. 200 Blasoperationen mehr, um den Zielzustand zu erreichen.

Beispiel 5**Eingabeparameter:**

```

Q = (2,2)
hof_size = (6,6)
use_binomial = False
tolerated_amount = 5
max_muster_operations = 5000
startwert = 100
choose_faster_path = True

```

Ausgabe in der Konsole:**AUSGANGSPUNKT:**

Verwendete Wahrscheinlichkeitsmodellierung: Erwartungswert-basiert

Laubverteilung vor dem Blasprozess:

100.0	100.0	100.0	100.0	100.0	100.0
100.0	100.0	100.0	100.0	100.0	100.0
100.0	100.0	100.0	100.0	100.0	100.0
100.0	100.0	100.0	100.0	100.0	100.0
100.0	100.0	100.0	100.0	100.0	100.0
100.0	100.0	100.0	100.0	100.0	100.0

Führe Simulation durch ...

ERGEBNIS DER SIMULATION:

Ausgeführte Blasoperationen: 437

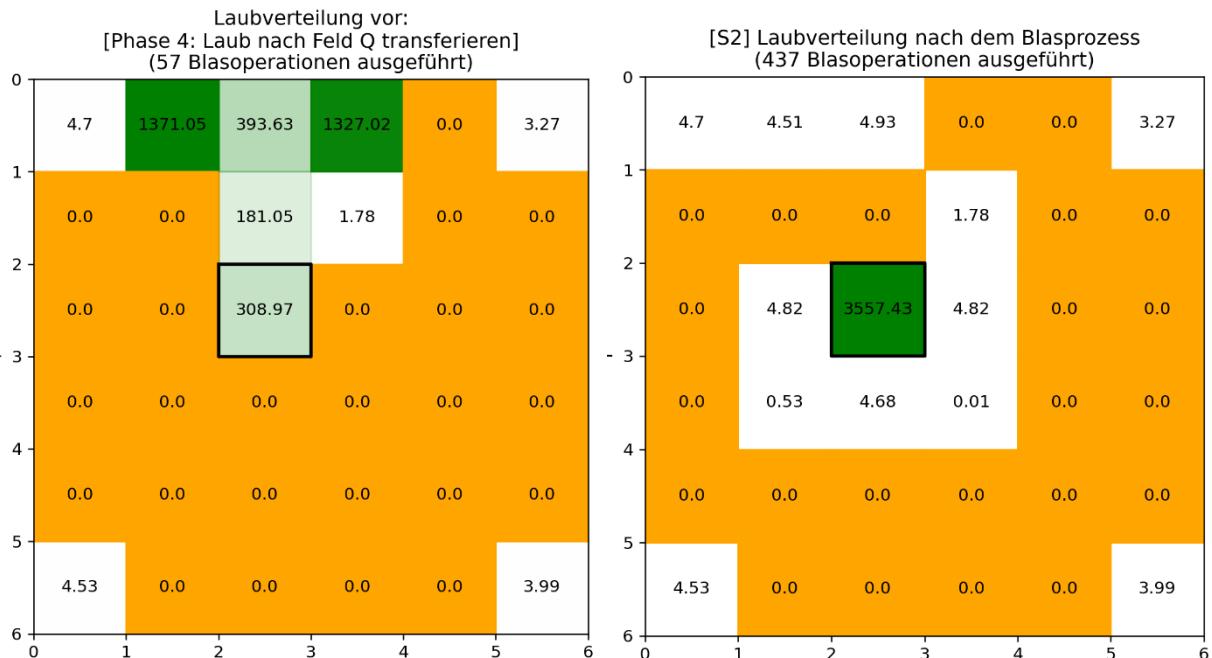
Anteil der Blätter auf Q an der Gesamtlaubmenge: 98.81743334969002 %

Laubverteilung nach dem Blasprozess:

4.6982	4.50521	4.925	0.0	0.0	3.2743
0.0	0.0	0.0	1.78055	0.0	0.0
0.0	4.82349	3557.4276	4.82349	0.0	0.0
0.0	0.5307	4.68416	0.01131	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0
4.52761	0.0	0.0	0.0	0.0	3.98838

GENERALISIERTER ABLAUFPLAN:

(...)

AusgabepLOTS:**Beobachtung:**

In diesem Beispiel wurden die Erwartungswerte zur Wahrscheinlichkeitsmodellierung verwendet. Dies sorgt dafür, dass gut sichtbar ist, welche Felder vollständig und welche nur asymptotisch geleert werden. An den im Plot angezeigten Zahlen lässt es sich zwar nicht immer zu ablesen, da diese gerundet werden. Allerdings werden nur die Felder, die vollständig leer sind, orange eingefärbt, die asymptotisch geleerten Felder sind weiß eingefärbt.

Beispiel 6**Eingabeparameter:** $Q = (1,1)$

hof_size = (3,4)

use_binomial = True

tolerated_amount = 5

max_muster_operations = 5000

```
startwert = 100
choose_faster_path = True
```

Ausgabe in der Konsole:**AUSGANGSPUNKT:**

Verwendete Wahrscheinlichkeitsmodellierung: binomial

Laubverteilung vor dem Blasprozess:

100		100		100
100		100		100
100		100		100
100		100		100

Führe Simulation durch ...

ERGEBNIS DER SIMULATION:

Ausgeführte Blasoperationen: 553

Anteil der Blätter auf Q an der Gesamtblaubmenge: 98.5 %

Laubverteilung nach dem Blasprozess:

3		0		5
0		1182		1
0		0		4
3		0		2

GENERALISIERTER ABLAUFPLAN:

```
[Phase 1: Unterste Reihe entlauben]
blase(Feld0: (0, 3), nach: oben)
blase(Feld0: (0, 2), nach: oben)
blase(Feld0: (0, 1), nach: oben)
Muster(Sourcefelder: [[(0, 3)]]) {
    blase(Feld0: (0, 2), nach: unten)
}
Muster(Sourcefelder: [[(0, 0)]]) {
    blase(Feld0: (0, 1), nach: oben)
}

[Phase 2: Laub auf oberste Reihe blasen]
blase(Feld0: (0, 3), nach: rechts)
blase(Feld0: (0, 2), nach: rechts)
blase(Feld0: (0, 1), nach: rechts)
blase(Feld0: (0, 0), nach: rechts)

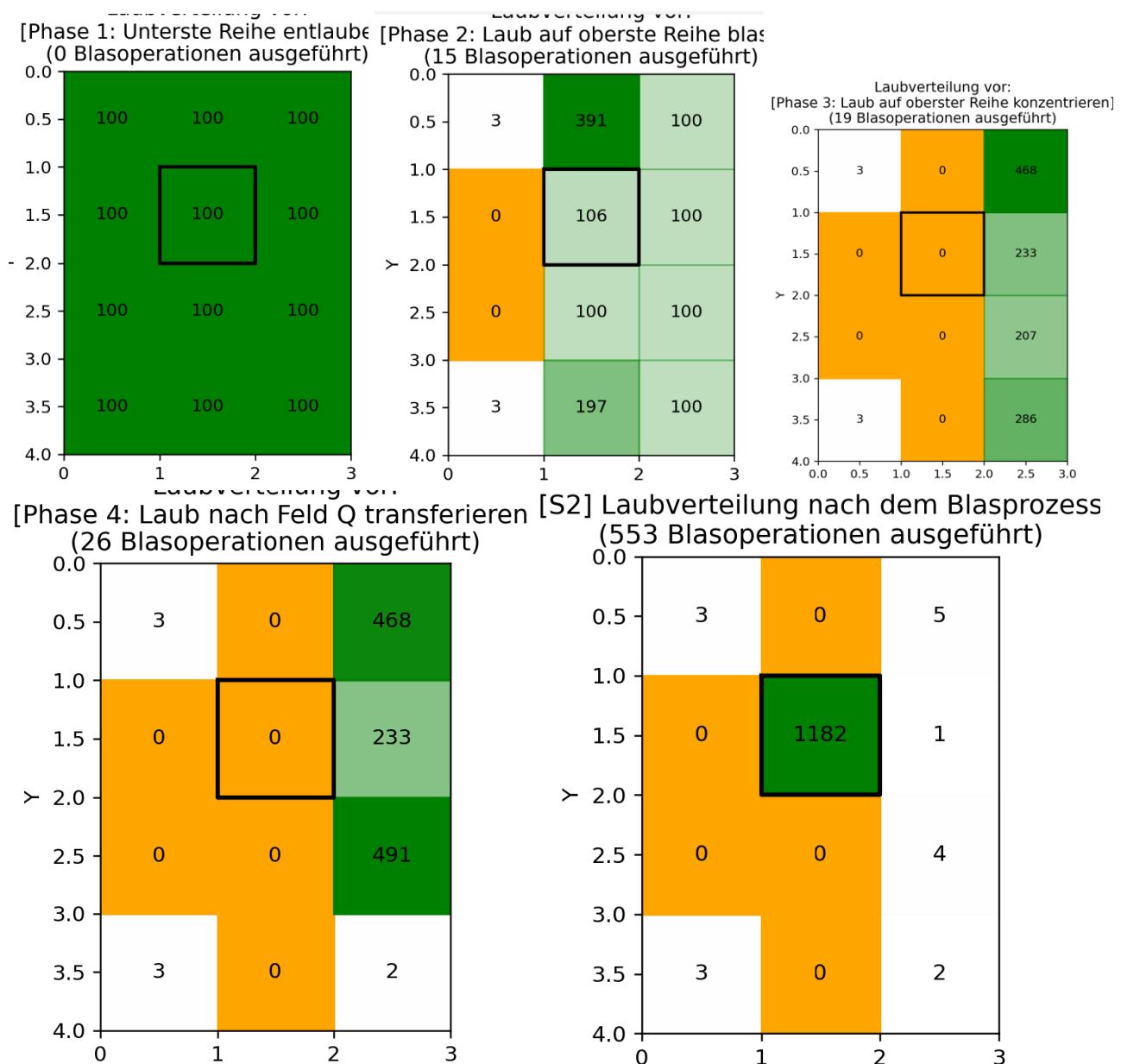
[Phase 3: Laub auf oberster Reihe konzentrieren]
Muster(Sourcefelder: [[(2, 3)]]) {
    blase(Feld0: (1, 3), nach: rechts)
}

[Phase 4: Laub nach Feld Q transferieren]
Muster(Sourcefelder: [[(2, 1), (2, 2), (2, 0)]]) {
    blase(Feld0: (1, 0), nach: rechts)
    blase(Feld0: (1, 1), nach: rechts)
    blase(Feld0: (2, 3), nach: oben)
}
```

Ausgabeplots:

Aufgabe 1: Laubmaschen

Teilnahme-ID: 69408



Beobachtung:

Auch bei sehr kleinen Höfen, die noch dazu unterschiedliche Seitenlängen haben, funktioniert das Programm einwandfrei.

Beispiel 7

Eingabeparameter:

```

Q = (8,9)
hof_size = (15,19)
use_binomial = False
tolerated_amount = 5
max_muster_operations = 5000
startwert = 100
choose_faster_path = True

```

Ausgabe in der Konsole:

Aufgabe 1: Laubmaschen

Teilnahme-ID: 69408

Ausgabe in der Konsole:

AUSGANGSPUNKT:

Verwendete Wahrscheinlichkeitsmodellierung: binomial

Laubverteilung vor dem Blasprozess:

Führe Simulation durch ...

ERGEBNIS DER SIMULATION:

Ausgeführte Blasoperationen: 981

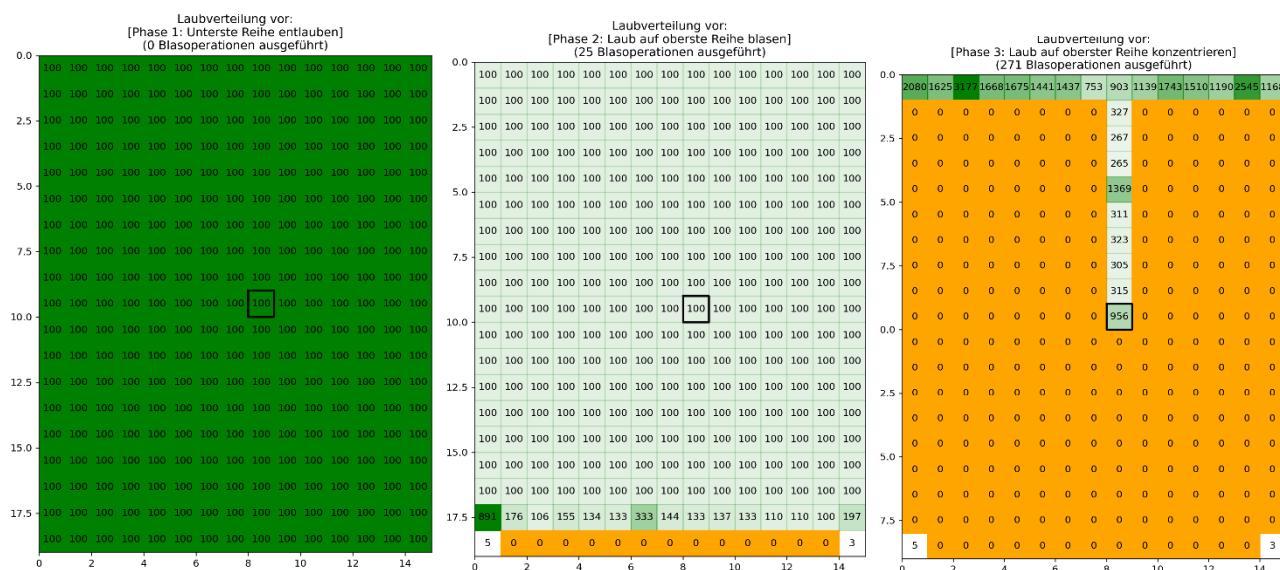
Anteil der Blätter auf 0 an der Gesamtblattmenge: 98.78947368421052 %

Anteil der Blätter auf Q an der Gesamtausbreitung nach dem Blasprozess:

GENERALISIERTER ABLAUFPLAN:

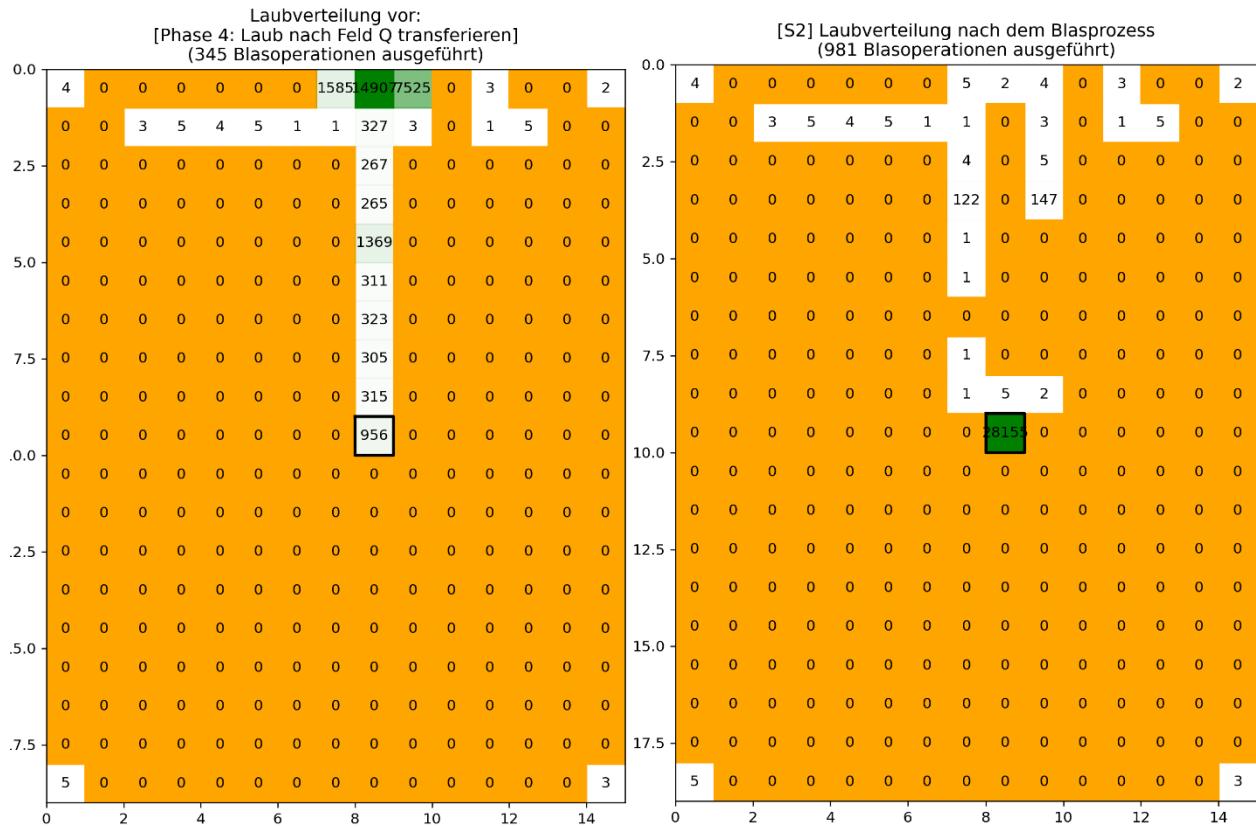
(. . .)

Ausgabeplots:



Aufgabe 1: Laubmaschen

Teilnahme-ID: 69408



Beobachtung:

Auch für große Höfe können trotzdem innerhalb von weniger als einer Sekunde sehr gute Ergebnisse gefunden werden. All dies macht deutlich, dass Ansatz 2 sowohl in Hinblick auf die Laufzeit als auch in Hinblick auf die Qualität weit besser als Ansatz 1 ist, der einen Hof dieser Größe gar nicht mehr lösen könnte.

Ansatz 3 (aufgabe3_solver.py)

Beispiel 1

Eingabeparameter:

$$Q = (2,2)$$

hof size = (5,5)

use binomial = True

binomial rank = “random”

initial_rank
startwert = 100

weight avg = 0.5

weight varianz = 0

max_operations = 1000

max_operations = 1000
satisfied_constraint = 0

satisfied_constraint

`max_operations` – tolerated amount =

tolerated_amount = 5
clear_edges = True

Math_Miscellaneous_operations

Ausgabe in der Ko

Verwendete Wahrscheinlichkeitsmodellierung: binomial

Laubverteilung vor dem Blasprozess:

100	100	100	100	100
100	100	100	100	100
100	100	100	100	100
100	100	100	100	100
100	100	100	100	100

Führe Simulation durch ...

ERGEBNIS DER SIMULATION:

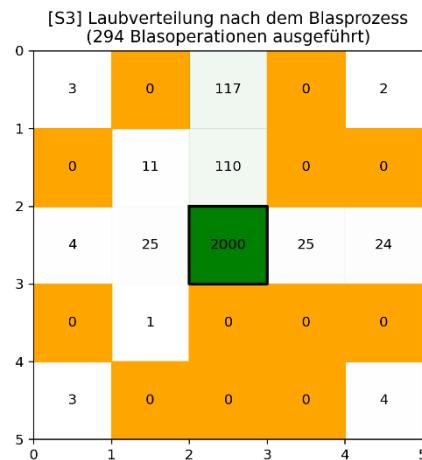
Ausgeführte Blasoperationen: 294

Anteil der Blätter auf Q an der Gesamtlaubmenge: 80.0 %

Laubverteilung nach dem Blasprozess:

3	0	117	0	2
0	11	110	0	0
4	25	2000	25	24
0	1	0	0	0
3	0	0	0	4

AusgabepLOTS:



Beobachtung:

Wie man sieht, kann Ansatz 3 etwas mehr Blätter als Ansatz 1 auf Feld Q befördern – das Ergebnis ist dennoch deutlich schlechter als das von Ansatz 2.

Beispiel 2

Eingabeparameter:

$Q = (1,1)$

$hof_size = (4,4)$

$use_binomial = True$

$binomial_rank = "random"$

$startwert = 100$

$weight_avg = 0.5$

$weight_varianz = 0.5$

$max_operations = 1000$

$satisfied_constraint = 0.8$

$max_operations = 1000$

```
tolerated_amount = 5
clear_edges = True
max_muster_operations = 1000
```

Ausgabe in der Konsole:

AUSGANGSPUNKT:

Verwendete Wahrscheinlichkeitsmodellierung: binomial

Laubverteilung vor dem Blasprozess:

100	100	100	100
100	100	100	100
100	100	100	100
100	100	100	100

Führe Simulation durch ...

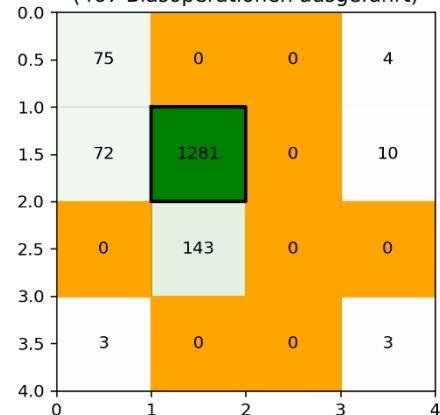
ERGEBNIS DER SIMULATION:

Ausgeführte Blasoperationen: 467

Anteil der Blätter auf Q an der Gesamtlaubmenge: 80.0625 %

Laubverteilung nach dem Blasprozess:

75	0	0	4
72	1281	0	10
0	143	0	0
3	0	0	3

AusgabepLOTS:[S3] Laubverteilung nach dem Blasproze
(467 Blasoperationen ausgeführt)

Beispiel 3

Eingabeparameter:

```
Q = (3,3)
hof_size = (4,4)
use_binomial = True
binomial_rank = "random"
startwert = 100
weight_avg = 0.5
weight_varianz = 0.5
max_operations = 1000
satisfied_constraint = 0.3
max_operations = 1000
```

```
tolerated_amount = 5
clear_edges = True
max_muster_operations = 1000
```

Ausgabe in der Konsole:**AUSGANGSPUNKT:**

Verwendete Wahrscheinlichkeitsmodellierung: binomial

Laubverteilung vor dem Blasprozess:

100	100	100	100	100	100
100	100	100	100	100	100
100	100	100	100	100	100
100	100	100	100	100	100
100	100	100	100	100	100
100	100	100	100	100	100

Führe Simulation durch ...

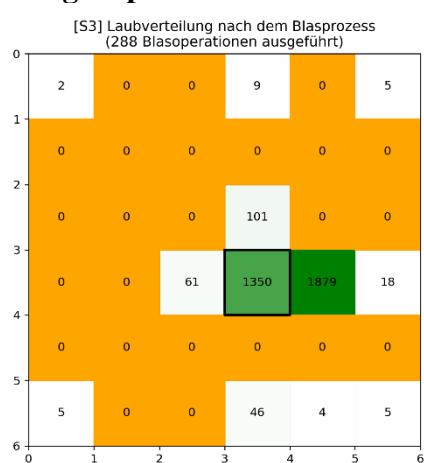
ERGEBNIS DER SIMULATION:

Ausgeföhrte Blasoperationen: 288

Anteil der Blätter auf Q an der Gesamtbläubmenge: 37.5 %

Laubverteilung nach dem Blasprozess:

2	0	0	9	0	5
0	0	0	0	0	0
0	0	0	101	0	0
0	0	61	1350	1879	18
0	0	0	0	0	0
5	0	0	46	4	5

Ausgabeplots:**Beispiele – Erweiterung 1: Keine Mauer-Umrandung**

Zum Demonstrieren der Funktionstüchtigkeit des Programms, das diese Erweiterung implementiert (Aufgabe 1 E1/aufgabe1_solver2.py) habe ich es ebenfalls mit verschiedenen Parametern ausgeführt. Die vollständigen Beispielausgaben sowie die vom Programm erstellten Dateien, die die durchgeföhrten Blasoperationen enthalten, sind im Ordner *Aufgabe 1 E1* zu finden.

Im Ordner Aufgabe 1 E1/Outputs sind für jedes Beispiel die Blasoperationen zu finden, die die Programme bei der Simulation durchgeführt haben. Sie sind in Textdateien gespeichert. Dabei sind die Blasoperationen, die von Beispiel x durchgeführt wurden, in der Textdatei „output_E1_{beispielx}.txt“ gespeichert.

Beispiel 1

Eingabeparameter:

$Q = (2,2)$
 $hof_size = (5,5)$
 $use_binomial = True$
 $tolerated_amount = 2$
 $max_muster_operations = 500$
 $startwert = 100$
 $choose_faster_path = True$

Ausgabe in der Konsole:

AUSGANGSPUNKT:

Verwendete Wahrscheinlichkeitsmodellierung: binomial

Laubverteilung vor dem Blasprozess:

100		100		100		100		100
100		100		100		100		100
100		100		100		100		100
100		100		100		100		100
100		100		100		100		100

Führe Simulation durch ...

ERGEBNIS DER SIMULATION:

Ausgeführte Blasoperationen: 455

Anteil der Blätter auf Q an der ursprünglichen Gesamtblaubmenge: 30.36 %

Gesamtblaubmenge, die sich noch auf dem Hof befindet: 779 von urspr. 2500

Laubverteilung nach dem Blasprozess:

0		1		0		0		0
0		0		2		0		0
0		8		759		2		2
0		0		0		2		1
0		2		0		0		0

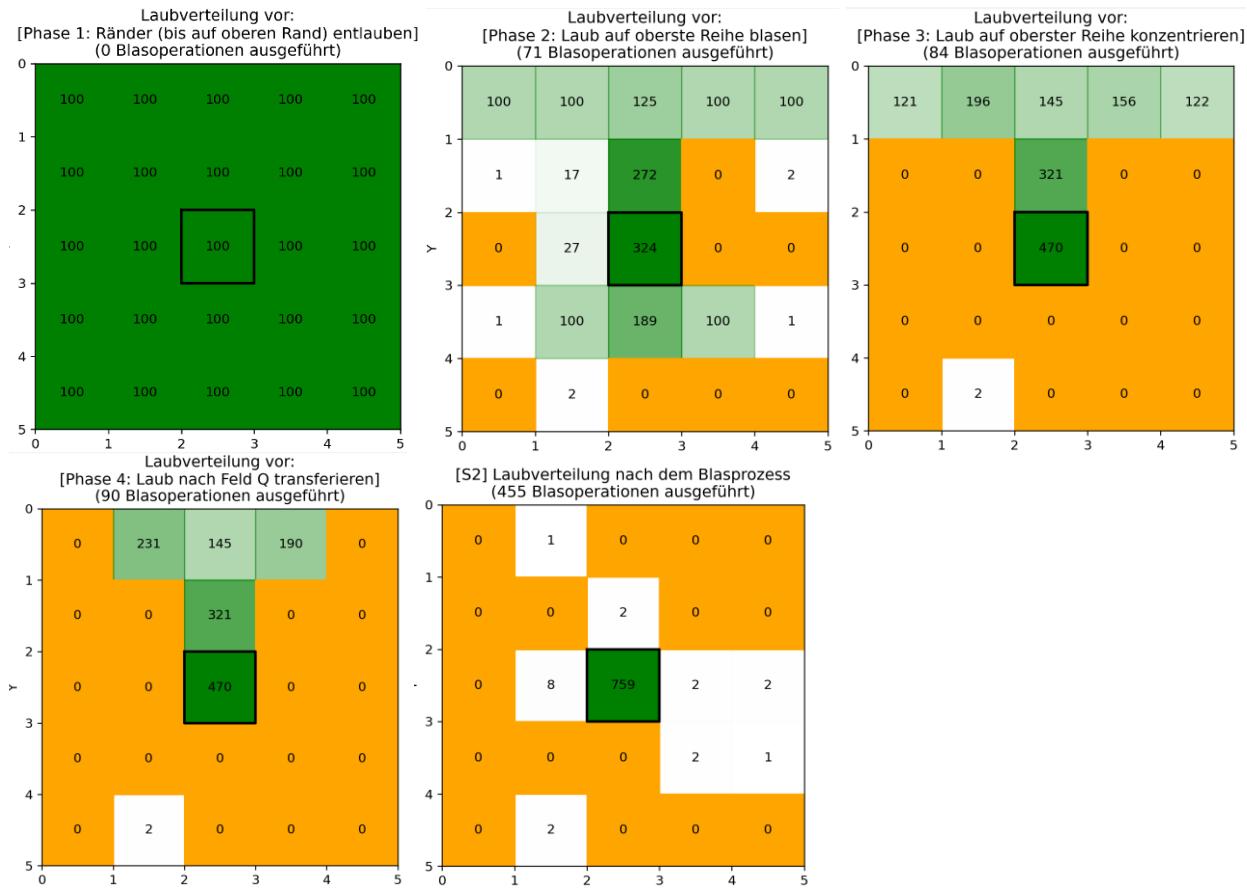
GENERALISIERTER ABLAUFPLAN:

(...)

Ausgabeplots:

Aufgabe 1: Laubmaschen

Teilnahme-ID: 69408



Beispiel 2

Eingabeparameter:

```
Q = (4,4)
hof_size = (10,10)
use_binomial = True
tolerated_amount = 2
max_muster_operations = 5000
startwert = 100
choose_faster_path = True
```

Ausgabe in der Konsole:

AUSGANGSPUNKT:

Verwendete Wahrscheinlichkeitsmodellierung: binomial

Laubverteilung vor dem Blasprozess:

100	100	100	100	100	100	100	100	100	100
100	100	100	100	100	100	100	100	100	100
100	100	100	100	100	100	100	100	100	100
100	100	100	100	100	100	100	100	100	100
100	100	100	100	100	100	100	100	100	100
100	100	100	100	100	100	100	100	100	100
100	100	100	100	100	100	100	100	100	100
100	100	100	100	100	100	100	100	100	100
100	100	100	100	100	100	100	100	100	100
100	100	100	100	100	100	100	100	100	100

Führe Simulation durch ...

ERGEBNIS DER SIMULATION:

Aufgabe 1: Laubmaschen

Teilnahme-ID: 69408

Ausgeführte Blasoperationen: 1153

Anteil der Blätter auf Q an der ursprünglichen Gesamtblaubmenge: 29.38 %

Gesamtblaubmenge, die sich noch auf dem Hof befindet: 2984 von urspr. 10000

Laubverteilung nach dem Blasprozess:

1	2	0	1	0	2	0	0	1	2
0	0	0	0	0	0	0	0	0	0
0	0	0	2	0	1	0	0	0	0
0	0	0	0	2	0	0	0	0	0
0	0	0	0	2938	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	23	0	8	0	0	0	1

GENERALISIERTER ABLAUFPLAN:

(...)

Ausgabeplots:



Beispiele – Erweiterung 2: Zwei Laubbläser

Zum Demonstrieren der Funktionstüchtigkeit des Programms, das diese Erweiterung implementiert (Aufgabe 1 E2/aufgabe1_solver2.py) habe ich es ebenfalls mit verschiedenen Parametern ausgeführt. Die vollständigen Beispielausgaben sowie die vom Programm erstellten Dateien, die die durchgeführten Blasoperationen enthalten, sind im Ordner *Aufgabe 1 E2* zu finden.

Im Ordner Aufgabe 1 E2/Outputs sind für jedes Beispiel die Blasoperationen zu finden, die die Programme bei der Simulation durchgeführt haben. Sie sind in Textdateien gespeichert. Dabei sind die Blasoperationen, die von Beispiel x durchgeführt wurden, in der Textdatei „output_E2_{beispielx}.txt“ gespeichert.

Beispiel 1

Eingabeparameter:

$Q = (2,2)$
 $hof_size = (5,5)$
 $use_binomial = \text{True}$
 $tolerated_amount = 5$
 $\max_muster_operations = 500$
 $\text{startwert} = 100$
 $\text{choose_faster_path} = \text{True}$

Ausgabe in der Konsole:

AUSGANGSPUNKT:

Verwendete Wahrscheinlichkeitsmodellierung: binomial

Laubverteilung vor dem Blasprozess:

100		100		100		100		100
100		100		100		100		100
100		100		100		100		100
100		100		100		100		100
100		100		100		100		100

Führe Simulation durch ...

ERGEBNIS DER SIMULATION:

Ausgeführte Blasoperationen: 59

Anteil der Blätter auf Q an der Gesamtbläubmenge: 96.04 %

Laubverteilung nach dem Blasprozess:

1		0		3		0		5
0		0		5		0		0
0		0		2401		0		0
0		0		29		0		0
2		50		0		0		4

GENERALISIERTER ABLAUFPLAN:

```
[Phase 1: Unterste Reihe entlauben]
blasen(
    BLÄSER1: Feld0: (0, 4), nach: rechts  BLÄSER2: Idle-Operation)
blasen(
    BLÄSER1: Feld0: (1, 4), nach: rechts  BLÄSER2: Idle-Operation)
blasen(
    BLÄSER1: Feld0: (2, 4), nach: rechts  BLÄSER2: Idle-Operation)
blasen(
    BLÄSER1: Feld0: (3, 4), nach: links  BLÄSER2: Feld0: (1, 4), nach: rechts)
blasen(
    BLÄSER1: Feld0: (2, 4), nach: links  BLÄSER2: Feld0: (2, 4), nach: rechts)
Muster(Sourcefelder: [[(0, 4), (4, 4)]] {
    blasen(
        BLÄSER1: Feld0: (1, 4), nach: links  BLÄSER2: Feld0: (3, 4), nach: rechts)
}
```

```
[Phase 2: Laub auf oberste Reihe blasen]
blasen(
    BLÄSER1: Feld0: (4, 4), nach: oben  BLÄSER2: Feld0: (3, 4), nach: oben)
```

Aufgabe 1: Laubmaschen

Teilnahme-ID: 69408

```

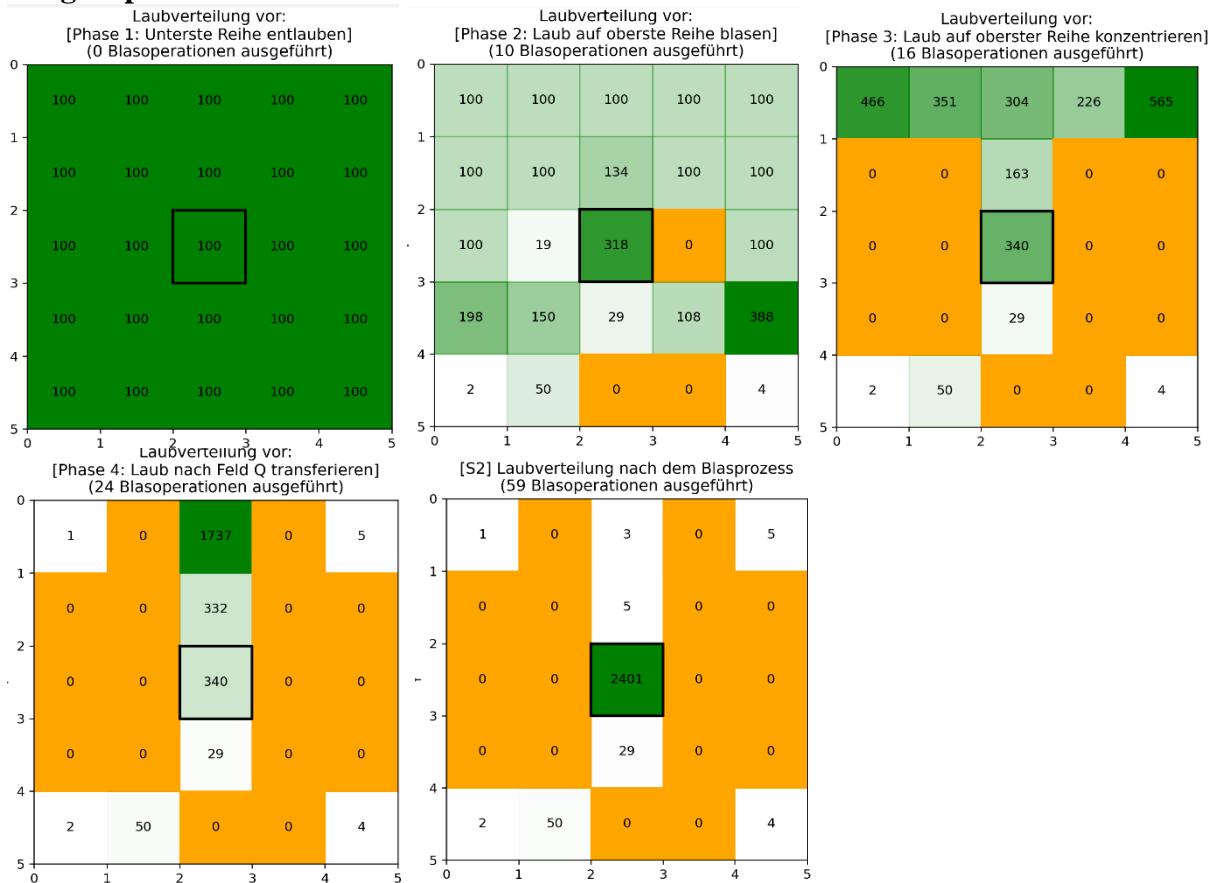
blase(
    BLÄSER1: Feld0: (1, 4), nach: oben   BLÄSER2: Feld0: (0, 4), nach: oben)
blase(
    BLÄSER1: Feld0: (4, 3), nach: oben   BLÄSER2: Feld0: (3, 3), nach: oben)
blase(
    BLÄSER1: Feld0: (1, 3), nach: oben   BLÄSER2: Feld0: (0, 3), nach: oben)
blase(
    BLÄSER1: Feld0: (4, 2), nach: oben   BLÄSER2: Feld0: (3, 2), nach: oben)
blase(
    BLÄSER1: Feld0: (1, 2), nach: oben   BLÄSER2: Feld0: (0, 2), nach: oben)

[Phase 3: Laub auf oberster Reihe konzentrieren]
Muster(Sourcefelder: [[(0, 0), (4, 0)]]) {
    blase(
        BLÄSER1: Feld0: (0, 1), nach: oben   BLÄSER2: Feld0: (4, 1), nach: oben)
}
blase(
    BLÄSER1: Feld0: (4, 0), nach: links   BLÄSER2: Feld0: (0, 0), nach: rechts)

[Phase 4: Laub nach Feld Q transferieren]
Muster(Sourcefelder: [[(2, 0), (2, 1)]]) {
    Muster(Sourcefelder: [[(2, 0)]]) {
        blase(
            BLÄSER1: Feld0: (1, 0), nach: rechts   BLÄSER2: Feld0: (3, 0), nach: links)
    }
    blase(
        BLÄSER1: Feld0: (1, 1), nach: rechts   BLÄSER2: Feld0: (3, 1), nach: links)
}

```

AusgabepLOTS:



Beispiel 2

Eingabeparameter:

$$Q = (5, 7)$$

$$hof_size = (15, 16)$$

Aufgabe 1: Laubmaschen

Teilnahme-ID: 69408

```
use_binomial = True  
tolerated_amount = 5  
max_muster_operations = 5000  
startwert = 100  
choose_faster_path = True
```

Ausgabe in der Konsole:

AUSGANGSPUNKT:
Verwendete Wahrscheinlichkeitsmodellierung: binomial

Führe Simulation durch ...

ERGEBNIS DER SIMULATION:

Ausgeführte Blasoperationen: 190

Anteil der Blätter auf 0 an der Gesamtlaubmenge: 99.90416666666667 %

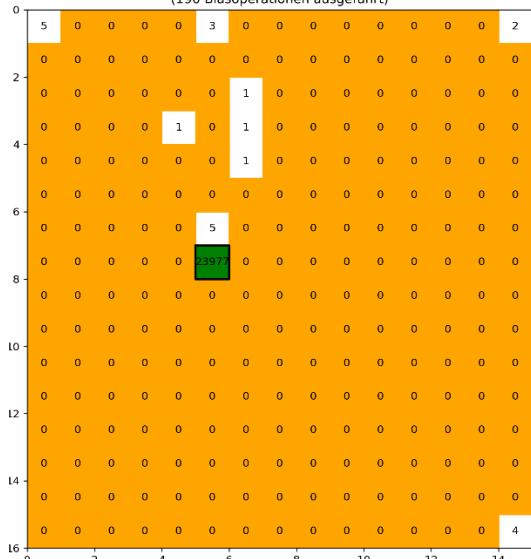
Anteil der Blätter an Q an der Gesamtausbreitung nach dem Blasprozess:

GENERALISIERTER ABLAUFPLAN:

(...,)

Ausgabeplots:

[S2] Laubverteilung nach dem Blasprozess
(190 Blasoperationen ausgeführt)



Beispiele – Erweiterung 3: Verschiedene Laubtypen

Zum Demonstrieren der Funktionstüchtigkeit des Programms, das diese Erweiterung implementiert (Aufgabe 1 E3/aufgabe1_solver2.py) habe ich es ebenfalls mit verschiedenen Parametern ausgeführt. Die vollständigen Beispielausgaben sowie die vom Programm erstellten Dateien, die die durchgeführten Blasoperationen enthalten, sind im Ordner *Aufgabe 1 E3* zu finden.

Im Ordner Aufgabe 1 E3/Outputs sind für jedes Beispiel die Blasoperationen zu finden, die die Programme bei der Simulation durchgeführt haben. Sie sind in Textdateien gespeichert. Dabei sind die Blasoperationen, die von Beispiel x durchgeführt wurden, in der Textdatei „output_E3_{beispielx}.txt“ gespeichert.

Beispiel 1

Eingabeparameter:

```
Q = (2,2)
hof_size = (5,5)
use_binomial = True
tolerated_amount = 5
max_muster_operations = 500
startwert = 100
choose_faster_path = True
```

Regeln, die für den zweiten Laubtyp gelten: Rules(

```
    use_binomial=use_binomial, binomial_rank="random",
    A_seitenabtrieb = 0.3, B_vorne_abtrieb = 0.3, A_noB_seitenabtrieb=0.5*1, s1=0.7, s4=0)
```

Für den ersten Laubtyp gelten die Standardregeln.

Ausgabe in der Konsole:

AUSGANGSPUNKT:

Verwendete Wahrscheinlichkeitsmodellierung: binomial

Laubverteilung vor dem Blasprozess:

100	100	100	100	100
100	100	100	100	100
100	100	100	100	100
100	100	100	100	100
100	100	100	100	100

Führe Simulation durch ...

ERGEBNIS DER SIMULATION:

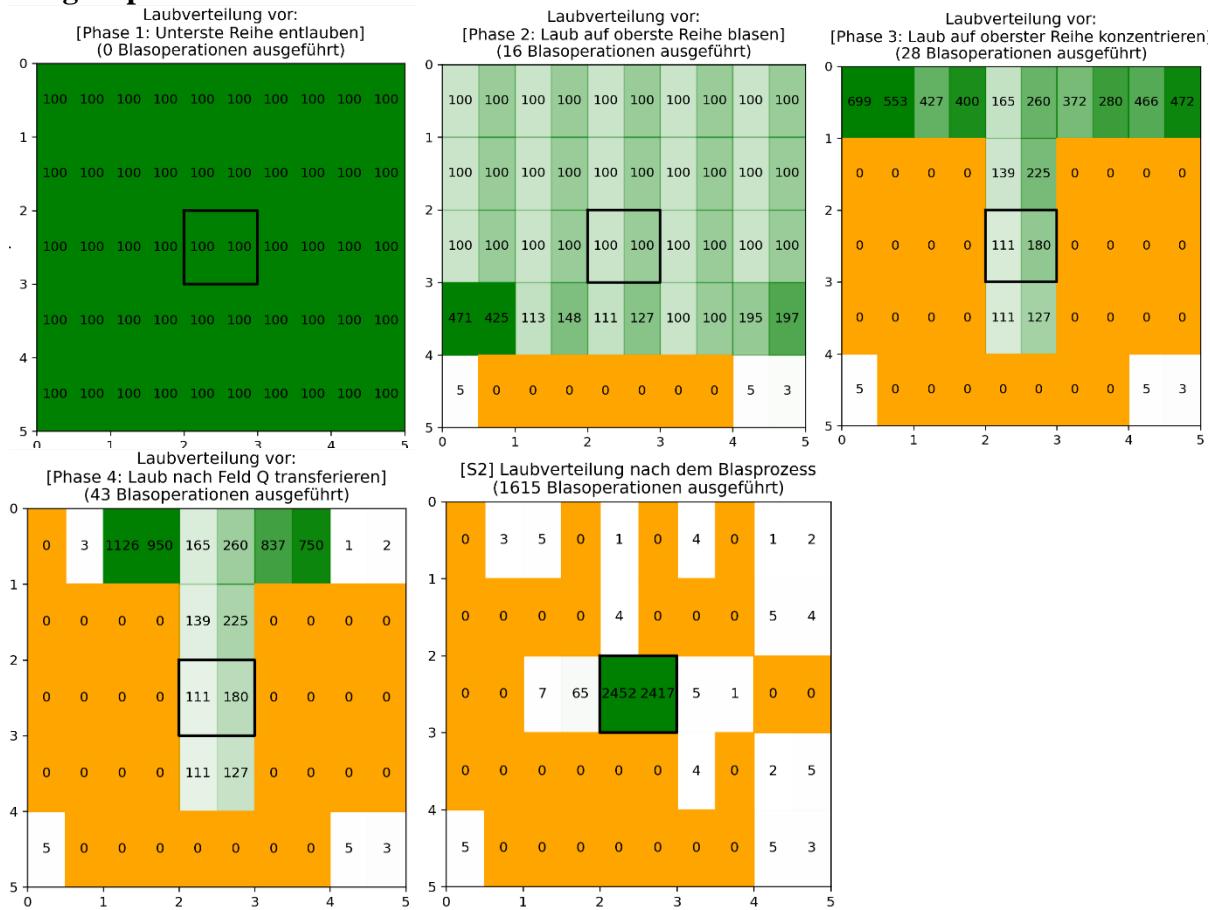
Ausgeführte Blasoperationen: 59

Anteil der Blätter auf Q an der Gesamtbläubmenge: 96.04 %

Laubverteilung nach dem Blasprozess:

1	0	3	0	5
0	0	5	0	0
0	0	2401	0	0
0	0	29	0	0
2	50	0	0	4

GENERALISIERTER ABLAUFPLAN:
(...)

AusgabepLOTS:**Beispiel 2****Eingabeparameter:** $Q = (1,2)$ $hof_size = (8,4)$ $use_binomial = True$ $tolerated_amount = 5$ $max_muster_operations = 500$ $startwert = 100$ $choose_faster_path = True$

Regeln, die für den zweiten Laubtyp gelten: Rules(

use_binomial=use_binomial, binomial_rank="random",

A_seitenabtrieb = 0.3, B_vorne_abtrieb = 0.3, A_noB_seitenabtrieb=0.5*1, s1=0.7, s4=0)

Für den ersten Laubtyp gelten die Standardregeln.

Ausgabe in der Konsole:

AUSGANGSPUNKT:

Verwendete Wahrscheinlichkeitsmodellierung: binomial

Laubverteilung vor dem Blasprozess:

100		100		100		100		100		100		100		100		100		100		100		100		100
100		100		100		100		100		100		100		100		100		100		100		100		100
100		100		100		100		100		100		100		100		100		100		100		100		100
100		100		100		100		100		100		100		100		100		100		100		100		100
100		100		100		100		100		100		100		100		100		100		100		100		100
100		100		100		100		100		100		100		100		100		100		100		100		100
100		100		100		100		100		100		100		100		100		100		100		100		100
100		100		100		100		100		100		100		100		100		100		100		100		100

Aufgabe 1: Laubmaschen

Teilnahme-ID: 69408

Führe Simulation durch ...

ERGEBNIS DER SIMULATION:

Ausgeführte Blasoperationen: 190

Anteil der Blätter auf Q an der Gesamtlaubmenge: 99.90416666666667 %

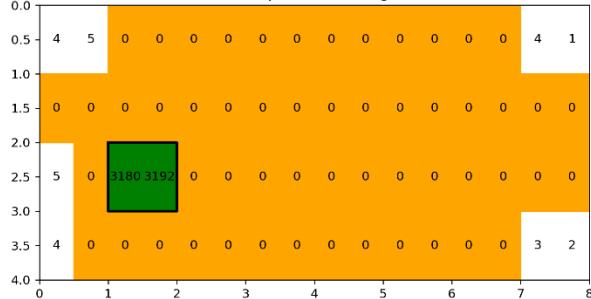
Laubverteilung nach dem Blasprozess:

GENERALISIERTER ABLAUFPLAN:

(...)

Ausgabeplots:

[S2] Laubverteilung nach dem Blasprozess
(561 Blasoperationen ausgeführt)



Quellcode

Im Folgenden einige Auszüge aus dem Quellcode meiner Implementierung.

Auszüge aus binomial util.py

In binomial_util.py sind Wahrscheinlichkeitsfunktionen definiert, die von allen Ansätzen / Erweiterungen verwendet werden.

```
def binomialpdf(*, n, p, k):
    """
    Returns:
        float: P(X=k) mit den Parametern n und p und einer
        binomialverteilte Größe X
    """
    # Diese Sonderfälle würden wegen der logarithmischen Implementierung der
    Benoulli-Formel für P(X=k) einen DomainError erzeugen und werden daher separat
    abgehandelt:
    if p == 1:
        return n == k
```

```

    elif p == 0:
        return n == 0      # Berechneter Wert wird gecached:
    bincomb = math.comb(n, k)
    if bincomb == 0:
        return 0 # Dieser Fall tritt auf, wenn k > n oder k < 0
    log_binom = math.log(bincomb) + k * math.log(p) + (n - k) * math.log(1 - p)
    return math.exp(log_binom)

def binomialdist(*, n, p, relevant_threshold=0.01):
    """
    Returns:
        list: P(x=k) für alle k und den Parametern n und p und einer
        binomialverteilten Größe X. Der Index in der Liste korrespondiert mit dem
        jeweiligen k-Wert
    """
    dist = np.zeros(n+1) # Mit Nullen gefüllte Liste erzeugen
    for k in range(math.floor(n*p),-1,-1):
        pdf = binomialpdf(n=n, p=p, k=k)
        dist[k] = pdf
        if pdf < relevant_threshold:
            break
    for k in range(math.ceil(n*p),n+1,1):
        pdf = binomialpdf(n=n, p=p, k=k)
        dist[k] = pdf
        if pdf < relevant_threshold: # der restliche Bereich ist
            vernachlässigbar, da die
            Wahrscheinlichkeiten verschwinden
            gering werden
            break
    if len(dist[dist==0]) != 0:
        fill_rest = (1- sum(dist)) / len(dist[dist==0])
        dist[dist==0] = fill_rest
    return dist

def binomial_likeliest(*, n, p, rank=0, handle_ties="higher"):
    """
    Bestimmt das k, für das P(X=k) maximal wird (also das k, das am
    wahrscheinlichsten Eintritt).
    Args:
        n (int), p (float): Parameter für die Binomialverteilung
        rank (int oder str): Gibt an, welches bzw. das wieviel-
    wahrscheinlichste k zurückgegeben werden soll. z.B. wird im Fall rank==0 das
    wahrscheinlichste k zurückgegeben.
        Wenn rank == "random", dann wird den tatsächlichen
    Wahrscheinlichkeiten entsprechend ein k zufällig ausgewählt
        handle_ties (str): Entweder "higher", "lower" oder "random". Gibt an,
    ob im Falle zweier gleich wahrscheinlicher Fälle (tritt auf wenn p=0.5 und
    floor(n/2) <= k <= ceil(n/2)) das größere oder kleinere k zurückgegeben werden
    soll (wenn handle_ties=="random", dann wird mit 50%-iger Wahrscheinlichkeit
    zufällig gewählt).
    Returns:
        int: Das gesuchte k
        float: Die Wahrscheinlichkeit, zu der k eintritt, bzw. P(X=k)
    
```

```
"""
if rank == "random":
    dist = binomialdist(n=n, p=p)
    k = random.choices(population=[i for i in range(n+1)], k=1,
    weights=dist)[0]
    return k, dist[k]
(...)
```

Auszüge aus hof.py

In hof.py ist eine Rules-Klasse und eineHof-Klasse definiert, die in allen Ansätzen für die Grundaufgabe und in den meisten Erweiterungen unverändert verwendet wird.

```
class Rules:
"""
Dient zum Speichern der Regeln, die auf einem Hof für den Laubblasprozess
gelten.
"""

def __init__(self, *, A_seitenabtrieb = 0.1, B_vorne_abtrieb = 0.1,
A_noB_seitenabtrieb=0.5*0.95, s1=0.9, s4=0.05, use_binomial=True,
binomial_rank="random", binomial_handle_ties="higher"):
    assert 0 < A_seitenabtrieb < 0.5
    assert 0 < B_vorne_abtrieb < 1
    assert 0 < A_noB_seitenabtrieb <= 0.5
    self.A_seitenabtrieb = A_seitenabtrieb
    self.B_vorne_abtrieb = B_vorne_abtrieb
    self.s1 = s1
    self.s4 = s4
    self.A_noB_seitenabtrieb = A_noB_seitenabtrieb

    assert isinstance(binomial_rank, int) or binomial_rank == "random"
    assert binomial_handle_ties == "higher" or binomial_handle_ties == "lower" or binomial_handle_ties == "random"
    self.use_binomial = use_binomial
    self.binomial_rank = binomial_rank
    self.binomial_handle_ties = binomial_handle_ties

class Hof:
"""
Repräsentiert einen aus Planquadraten bestehenden Hof, auf dem Laub
geblasen werden kann
"""

def __init__(self, size, rules = Rules(), *, startwert=100, felder=None,
blas_counter=0):
    self.x_size = size[0] # Hof quadratisch -> x_size und y_size sind
    # gleich
    self.y_size = size[1]
    self.rules = rules
    self.startwert = startwert

    if felder is None:
```

```

        self.felder = np.full((self.x_size, self.y_size), startwert,
                               dtype=int if self.rules.use_binomial else float)
    else:
        self.felder = np.array(felder, dtype=int if self.rules.use_binomial
else float)
    self.blae_counter = blaे_counter
    self.blae_log = []

def blaе(self, feld0, blow_direction):
    """
    Simuliert einen Blasvorgang und aktualisiert self.felder auf die
    resultierende Blattverteilung.

    Args:
        feld0 (int): Index des Felds, auf dem der Hausmeister steht
        blow_direction (tuple): Richtung, in die der Hausmeister bläst.
            Kann folgende Werte annehmen: (1,0) (=rechts), (-1,0) (=links),
            (0,1) (=unten), (0,-1) (=oben)
    """
    if not blow_direction in [(0,1),(0,-1),(1,0),(-1,0)]:
        return
    self.blae_counter += 1
    self.blae_log.append(dict(feld0=feld0, blow_direction=blow_direction))

    # Richtung, die orthogonal zur Blasrichtung ist, ermitteln:
    orthogonal_direction = self.get_orthogonal_direction(blown_direction)
    # Feld A (Feld unmittelbar vor dem Laubbläser) ermitteln:
    feldA = (feld0[0]+blown_direction[0], feld0[1]+blown_direction[1])
    if not self.does_exist(feldA):
        return
    new_feldA_value = 0

    # Feld B (Feld hinter Feld A) ermitteln:
    feldB = (feldA[0]+blown_direction[0], feldA[1]+blown_direction[1])
    if self.does_exist(feldB):
        # -> Es gibt ein Feld B
        if self.rules.use_binomial:

            A_seitenabtrieb_1, _ = binomial_likeiest(n=self.felder[feldA],
p=self.rules.A_seitenabtrieb, rank=self.rules.binomial_rank,
handle_ties=self.rules.binomial_handle_ties)
            A_seitenabtrieb_2, _ = binomial_likeiest(n=self.felder[feldA]-
A_seitenabtrieb_1, p=self.rules.A_seitenabtrieb/(1-self.rules.A_seitenabtrieb),
rank=self.rules.binomial_rank, handle_ties=self.rules.binomial_handle_ties)
            B_vorne_abtrieb, _ = binomial_likeiest(n=self.felder[feldB],
p=self.rules.B_vorne_abtrieb, rank=self.rules.binomial_rank,
handle_ties=self.rules.binomial_handle_ties) # Anzahl an Blättern von Feld B,
die nach vorne abgetrieben wurden

        else:
            A_seitenabtrieb_1 = self.felder[feldA] *
                self.rules.A_seitenabtrieb
            A_seitenabtrieb_2 = A_seitenabtrieb_1

```

```

        B_vorne_abtrieb = self.felder[feldB] *
            self.rules.B_vorne_abtrieb
        new_feldB_value = self.felder[feldA] - (A_seitenabtrieb_1 +
A_seitenabtrieb_2) + self.felder[feldB] - B_vorne_abtrieb

        # Nachbarfeld von Feld B, das Feld A gegenüberliegt, aktualisieren
        # (sofern vorhanden):
        if self.does_exist((feldB[0]+blow_direction[0],
            feldB[1]+blow_direction[1])):
            B_seitenabtrieb_1 = 0
            B_seitenabtrieb_2 = 0
            self.felder[(feldB[0]+blow_direction[0],
                feldB[1]+blow_direction[1])] += B_vorne_abtrieb
        else:
            if self.rules.use_binomial:

                B_seitenabtrieb_1, _ =
binomial_likeliest(n=B_vorne_abtrieb, p=(1-self.rules.s1)/2,
rank=self.rules.binomial_rank, handle_ties=self.rules.binomial_handle_ties)
                B_seitenabtrieb_2, _ =
binomial_likeliest(n=B_vorne_abtrieb-B_seitenabtrieb_1, p=((1-
self.rules.s1)/2)/(1-(1-self.rules.s1)/2), rank=self.rules.binomial_rank,
handle_ties=self.rules.binomial_handle_ties)
                new_feldB_value += B_vorne_abtrieb - B_seitenabtrieb_1 -
B_seitenabtrieb_2
            else:
                new_feldB_value += B_vorne_abtrieb * self.rules.s1
                B_seitenabtrieb_1 =
                    B_vorne_abtrieb * (1-self.rules.s1) * 0.5
                B_seitenabtrieb_2 = B_seitenabtrieb_1

        # Nachbarfelder von Feld B, die Feld A nicht gegenüberliegen,
        # aktualisieren (sofern vorhanden):
        if self.does_exist((feldB[0]+orthogonal_direction[0],
            feldB[1]+orthogonal_direction[1])):
            self.felder[(feldB[0]+orthogonal_direction[0],
                feldB[1]+orthogonal_direction[1])]
            ] += A_seitenabtrieb_1 + B_seitenabtrieb_1
        else:
            new_feldB_value += A_seitenabtrieb_1 + B_seitenabtrieb_1
        if self.does_exist((feldB[0]-orthogonal_direction[0], feldB[1]-
orthogonal_direction[1])):
            self.felder[(feldB[0]-orthogonal_direction[0], feldB[1]-
orthogonal_direction[1])]
            ] += A_seitenabtrieb_2 + B_seitenabtrieb_2
        else:
            new_feldB_value += A_seitenabtrieb_2 + B_seitenabtrieb_2

        # Feld B aktualisieren
        self.felder[feldB] = new_feldB_value
    else:
        # -> Es gibt kein Feld B.

```

```

if self.rules.use_binomial:

    A_noB_seitenabtrieb_1, _ =
binomial_likeyest(n=self.felder[feldA], p=self.rules.A_noB_seitenabtrieb,
rank=self.rules.binomial_rank, handle_ties=self.rules.binomial_handle_ties)
    A_noB_seitenabtrieb_2, _ =
binomial_likeyest(n=self.felder[feldA]-A_noB_seitenabtrieb_1,
p=self.rules.A_noB_seitenabtrieb/(1-self.rules.A_noB_seitenabtrieb),
rank=self.rules.binomial_rank, handle_ties=self.rules.binomial_handle_ties)
else:
    A_noB_seitenabtrieb_1 = self.felder[feldA] *
        self.rules.A_noB_seitenabtrieb
    A_noB_seitenabtrieb_2 = A_noB_seitenabtrieb_1
new_feldA_value = self.felder[feldA] - (A_noB_seitenabtrieb_1 +
    A_noB_seitenabtrieb_2)

# Nachbarfelder von Feld A, die dem laubblasenden Hausmeister nicht
gegenüberliegen, aktualisieren, sofern vorhanden:
if self.does_exist((feldA[0]+orthogonal_direction[0],
feldA[1]+orthogonal_direction[1])):
    self.felder[(feldA[0]+orthogonal_direction[0],
feldA[1]+orthogonal_direction[1])] += A_noB_seitenabtrieb_1
else:
    # -> Feld A ist ein Eckfeld, ein Teil des Laubs verbleibt also
    auf Feld A.
    new_feldA_value += A_noB_seitenabtrieb_1
if self.does_exist((feldA[0]-orthogonal_direction[0], feldA[1]-
orthogonal_direction[1])):
    self.felder[(feldA[0]-orthogonal_direction[0], feldA[1]-
orthogonal_direction[1])] += A_noB_seitenabtrieb_2
else:
    # -> Feld A ist ein Eckfeld, ein Teil des Laubs verbleibt also
    auf Feld A.
    new_feldA_value += A_noB_seitenabtrieb_2

# Feld A aktualisieren:
self.felder[feldA] = new_feldA_value

```

Grundaufgabe: Ansatz 1

Hilfsfunktionen zur Berechnung von Standardabweichung und Manhattan-Distanz:

```

def squared_std(values : list[float]):
    """
    Returns:
        float: Die quadrierte Standardabweichung (= Varianz) der Werte
        in values
    """
    mean = sum(values) / len(values)
    squared_diff = [(value - mean) ** 2 for value in values]
    return sum(squared_diff)

```

```
def manhattan_distance(feld0, feld1) -> int:
    """
    Returns:
        int: Die Manhatten-Distanz zwischen den Tupeln feld0 und feld1,
              definiert als  $d = (feld1[0] - feld0[0]) + (feld1[1] - feld0[1])$ 
    """
    return abs(feld1[0] - feld0[0]) + abs(feld1[1] - feld0[1])
```

Klasse Solver1, die Methoden zum Durchführen des Blasprozesses (wie z.B. den Greedy-Algorithmus und Methoden zum Berechnen der Heuristiken) beinhaltet:

```
class Solver1:
    def __init__(self, hof, *, satisfied_constraint, max_operations=1000, Q, weight_avg, weight_varianz):
        """
        Args:
            hof (Hof): Der Hof, auf den sich die Strategie beziehen soll
            max_operations (int): Anzahl an Blasoperationen, nach der die
                                  Strategie auf jeden Fall abbricht
            satisfied_constraint (float): Anteil des Laubs auf Q am
                                          Gesamtblaub, nach dessen Erreichen die Strategie auf jeden
                                          Fall abbricht
            Q (tuple): Der Index von Feld Q
            weight_avg (float), weight_varianz (float): Die Gewichte, mit
                                              denen die Größen durchschnittl. Blattabstand zu Q und
                                              Varianz der Blattabstände zu Q in der Greedy Heuristik
                                              gewichtet bzw. multipliziert werden
        """
        self.hof = hof
        self.max_operations = max_operations
        self.satisfied_constraint = satisfied_constraint
        self.Q = Q
        if not self.hof.does_exist(Q):
            print("Q existiert nicht.")
            exit()
        if self.hof.is_edge(Q):
            print("Q darf kein Rand-/Eckfeld sein.")
            exit()
        self.clear_edges = True
        self.sum_laub = np.sum(self.hof.felder)
        # Gesamtanzahl an Blättern im Schulhof / im System bestimmen
        self.weight_avg = weight_avg
        self.weight_varianz = weight_varianz

        # Variablen, die den Fortschritt beim Leeren der Ränder speichern:
        self.edge_fields_to_clear = [
            (x, 0) for x in range(self.hof.x_size)] + [
            (x, self.hof.y_size-1) for x in range(self.hof.x_size)] + [
            (0, y) for y in range(self.hof.y_size)] + [
            (self.hof.x_size-1, y) for y in range(self.hof.y_size)]

        possible_edge_target_fields = []
        if not self.hof.y_size == 3:
            possible_edge_target_fields += [(0, self.Q[1]), (self.hof.x_size-1, self.Q[1])]
        if not self.hof.x_size == 3:
```

```

possible_edge_target_fields += [(self.Q[0],0),(self.Q[0],self.hof.y_size-1)]
if possible_edge_target_fields == []:
    self.edge_target_field = None
else:
    self.edge_target_field = min(
        possible_edge_target_fields, key=lambda x : manhattan_distance(x, self.Q)
    )
self.clear_edge_last_field_index = 0 # Im Prozess des Verschiebens von Laub vom Feldrand
                                   # auf ein Nicht-Randfeld (Teil der self.greedy_edge
                                   # Methode) wird hier die zuletzt durchgeführte
                                   # Operation gespeichert
self.edge_target_field_neighbors = [] # Nachbarfelder das "Rand-Zielfelds"

```

Hilfsfunktion, die die Randdistanz zwischen zwei Randfeldern bestimmt (gehört zur Solver1-Klasse):

```

def edge_distance(self, feld0, feld1):
    """
    returns:
        int: Die kleinste Verbindungsstrecke zwischen den Randfeldern
             feld0 und feld1, die nur über Rand- und Eckfelder läuft
    """
    if feld0[0] == 0 or feld0[0] == self.hof.x_size -1:
        # -> Feld 0 liegt auf unterem Rand
        if feld0[0] == feld1[0]:
            # -> Feld 0 und Feld 1 liegen in derselben "Zeile" - in
            # diesem Fall ist der gesuchte Abstand der x-
            # Koordinatenunterschied zwischen den Feldern
            return abs(feld0[1] - feld1[1])
        # Idee: Die beiden Ecken ermitteln, die zu Feld0 den
        # geringsten Abstand haben.
        # min(distance(feld0, c) + distance(c, feld1)) entspricht
        # nämlich (für das Eckfeld c, für das der Ausdruck minimal
        # wird) dem gesuchten Abstand zwischen Feld 0 und Feld 1.
        c1 = (feld0[0],0)
        d1 = feld0[1]
        c2 = (feld0[0],self.hof.y_size-1)
        d2 = abs(self.hof.y_size-1-feld0[1])
    else:
        if feld0[1] == feld1[1]:
            return abs(feld0[0] - feld1[0])
        c1 = (0,feld0[1])
        d1 = feld0[0]
        c2 = (self.hof.x_size-1,feld0[1])
        d2 = abs(self.hof.x_size-1-feld0[0])
    return
min([d1+manhattan_distance(c1,feld1),d2+manhattan_distance(c2,feld1)])

```

Greedy-Algorithmus, der auf Randfelder angewendet wird (gehört zur Solver1-Klasse):

Hinweis: Zur Verbesserung der Lesbarkeit des Dokuments wurden sämtliche Kommentare aus dem Quellcode entfernt. Es wird daher empfohlen, den Quellcode (aufgabe1_solver1.py) in einer geeigneten IDE zu öffnen und die Funktion dort zu lesen.

```

def greedy_edges(self):
    """
    Findet die beste Blasoperation zum Leeren des Rands (systematisches Verfahren)
    """

    if self.edge_target_field is None:
        return None
    if self.edge_fields_to_clear == []:
        return None
    possible_blow_directions = [(0,1),(1,0),(0,-1),(-1,0)]
    field_to_clear = max(self.edge_fields_to_clear, key=lambda x : self.edge_distance(x, self.edge_target_field))

    if self.hof.felder[field_to_clear] > self.hof.startwert * (1-self.satisfied_constraint):
        if manhattan_distance(field_to_clear, self.edge_target_field) == 1:
            if not field_to_clear in self.edge_target_field_neighbors:
                self.edge_target_field_neighbors.append(field_to_clear)

    elif field_to_clear == self.edge_target_field:
        self.edge_fields_to_clear += self.edge_target_field_neighbors
        if self.clear_edge_last_field_index == 0:
            self.clear_edge_last_field_index = 1
        elif self.clear_edge_last_field_index == 1:
            self.clear_edge_last_field_index = 2
        if self.edge_target_field[0] == 0:
            return dict(feld0=(1,self.edge_target_field[1]), blow_direction=(-1,0))
        elif self.edge_target_field[0] == self.hof.x_size-1:
            return dict(feld0=(self.hof.x_size-2,self.edge_target_field[1]),
blow_direction=(1,0))
        elif self.edge_target_field[1] == 0:
            return dict(feld0=(self.edge_target_field[0],1), blow_direction=(0,-1))
        elif self.edge_target_field[1] == self.hof.y_size-1:
            return dict(feld0=(self.edge_target_field[0],self.hof.y_size-2),
blow_direction=(0,1))
        else:
            self.clear_edge_last_field_index = 0
        field_to_clear = self.edge_target_field_neighbors[self.clear_edge_last_field_index]

    for blow_direction in possible_blow_directions:
        feld0=(field_to_clear[0]-blow_direction[0],field_to_clear[1]-blow_direction[1])
        if not self.hof.does_exist(feld0):
            continue
        if self.hof.is_corner(field_to_clear):
            orthogonal_direction = self.hof.get_orthogonal_direction(blow_direction)
            target_field =
(field_to_clear[0]+orthogonal_direction[0],field_to_clear[1]+orthogonal_direction[1])
            if not (self.hof.is_edge(target_field) and target_field in
self.edge_fields_to_clear):
                target_field =
(field_to_clear[0]-orthogonal_direction[0],field_to_clear[1]-
orthogonal_direction[1])
                if not (self.hof.is_edge(target_field) and target_field in
self.edge_fields_to_clear):
                    continue
            else:
                if field_to_clear in self.edge_fields_to_clear:
                    self.edge_fields_to_clear.remove(field_to_clear)
                target_field =
(field_to_clear[0]+blow_direction[0],field_to_clear[1]+blow_direction[1])
                if not self.hof.does_exist(target_field):
                    continue
                if self.edge_distance(target_field, self.edge_target_field) <
self.edge_distance(field_to_clear, self.edge_target_field):
                    return dict(

```

```

        feld0=feld0,
        blow_direction=blow_direction
    )
else:
    self.edge_fields_to_clear.remove(field_to_clear)
return self.greedy_edges()

```

Greedy-Algorithmus, der auf Nicht-Randfelder angewendet wird (gehört zur Solver1-Klasse):

```

def greedy_mid(self, *, weight_varianz, weight_avg):
    """
    Findet die beste Blasoperation zum Verschieben von Laub von Nicht-Randfeldern Richtung Q.
    (unter Verwendung einer Greedy-Heuristik)
    Returns:
        dict: Die als nächstes auszuführende Blasoperation.
    """
    possible_blow_directions = [(0,1),(1,0),(0,-1),(-1,0)]
    blattdistanzen = self.blattdistanzen(self.hof, self.Q, ignore_edge=False)
    # Berechnet alle Blattdistanzen vom Feld Q aus
    current_mw_bd = sum(blattdistanzen) / len(blattdistanzen)
    # Die mittlere Blattdistanz am aktuellen Hof
    best_score = float("inf")
    # Bester (= niedrigste) Wert für die STABW der Blattdistanz an einem Hof, der aus einer
    # möglichen Blasoperation resultiert
    best_op = None # Die Blasoperation, aus der der best_varianz_bd Wert resultiert
    for x in range(0,self.hof.x_size):
        for y in range(0,self.hof.y_size):
            for blow_direction in possible_blow_directions:
                if not self.hof.is_edge((x,y)):
                    if self.hof.is_edge((x+blow_direction[0],y+blow_direction[1])) or
self.hof.is_edge((x+2*blow_direction[0],y+2*blow_direction[1])):
                        continue # Laub auf Rand blasen ist verboten
                    if not manhattan_distance((x,y), self.Q) <
manhattan_distance((x+blow_direction[0], y+blow_direction[1]), self.Q):
                        hof_copy = self.hof.copy()
                        hof_copy.rules.use_binomial=False # Beim Finden der besten Blasoperation
                                            # mit den Erwartungswerten arbeiten, um
                                            # die Zufallskomponente zu entfernen
                        hof_copy.blase((x,y), blow_direction)
                        blattdistanzen = self.blattdistanzen(hof_copy, self.Q, ignore_edge=False)
                        if sum(blattdistanzen) / len(blattdistanzen) < current_mw_bd:
                            # Erstes, priorisiertes Maß der Heuristik. Nur Operationen, die
                            # dieses Maß verringern, werden zugelassen
                            score = squared_std(blattdistanzen) * weight_varianz +
(sum(blattdistanzen) / len(blattdistanzen)) * weight_avg # Gewichtetes Produkt aus erstem und
zweitem Maß bilden
                            if score < best_score:
                                # Blasoperation mit dem besten Score finden
                                best_score = score
                                best_op = dict(
feld0=tuple((x,y)), blow_direction=tuple(blow_direction)
)
    return best_op

```

step-Funktion, die bei Aufruf die Greedy-Heuristik ausführt, um die als nächstes auszuführende Blasoperation zu bestimmen, und diese anschließend ausführt:

```

def step(self):
    """
    Führt den nächsten Schritt des Blasprozesses aus:
    Mithilfe von self.greedy wird die nächste Blasoperation bestimmt, die anschließend
    ausgeführt wird
    """

```

```

if self.clear_edges:
    next_op = self.greedy_edges()
else:
    next_op = self.greedy_mid(weight_varianz=self.weight_varianz,
weight_avg=self.weight_avg)
    if next_op is None:
        self.clear_edges = not self.clear_edges
        if self.clear_edges:
            self.edge_fields_to_clear = [(x,0) for x in range(self.hof.x_size)] +
[(x,self.hof.y_size-1) for x in range(self.hof.x_size)] + [(0,y) for y in range(self.hof.y_size)] +
[(self.hof.x_size-1,y) for y in range(self.hof.y_size)]
            self.edge_target_field = min([(0,0),(1,0),(0,1),(1,1)], key=lambda x : manhattan_distance(x, self.Q))
    next_op = self.greedy_edges()
else:
    next_op = self.greedy_mid(weight_varianz=self.weight_varianz,
weight_avg=self.weight_avg)
    if next_op is not None:
        return True
else:
    self.hof.blase(next_op["feld0"], next_op["blow_direction"])
    if self.satisfied_constraint is not None:
        if self.hof.felder[self.Q] / self.sum_laub >= self.satisfied_constraint:
            return False
    if self.max_operations is not None:
        if self.hof.blas_counter >= self.max_operations:
            return False
return True

```

Grundaufgabe: Ansatz 2

Klasse zur Implementierung des Muster-Konzepts:

```

class Muster:
    """
    Implementierung des theoretischen Konzept eines Musters
    (siehe Dokumentation)
    """

    def __init__(self, strategy, source_fields, operations, tolerated_amount : float, *, num_max_operations=None):
        self.strategy = strategy # Die Strategie, zu der das Muster gehört
        self.hof = self.strategy.hof
        if num_max_operations is None:
            self.num_max_operations = strategy.max_muster_operations
        else:
            self.num_max_operations = num_max_operations
        self.source_fields = [
            strategy.rotate_field_index(f) for f in source_fields]
        self.operations = [
            strategy.rotate_blasoperation(o) if isinstance(o, dict) else o for
            o in operations]
        self.tolerated_amount = tolerated_amount
        self.reset()

    def reset(self):
        """
        Zurücksetzen des Musters auf den Zustand vor seiner Ausführung

```

```

    """
self.num_operations = 0 # Bisher durchgeführte Blasoperationen
self.check_for_changes = not (self.hof.rules.use_binomial is True and
                             self.hof.rules.binomial_rank == "random")
if self.check_for_changes:
    # Es wird nach jedem vollständigen Musterdurchlauf überprüft, ob
    # nach Veränderungen an der Gesamtblaubmenge auf den source-Feldern
    # stattfinden
    # Diese Überprüfung findet aber nicht statt, wenn die
    # Laubblassimulation den tatsächlichen Zufall simuliert
    self.current_sum = 0
self.next_op_index = 0 # Index der als nächst. auszuführenden Operation

def step(self):
    """
    Führt basierend auf self.operations die nächste Blasoperation aus
    """
    if isinstance(self.operations[self.next_op_index], dict):
        self.next_op_index += 1
        self.num_operations += 1

    elif isinstance(self.operations[self.next_op_index], Muster):
        # -> Ein anderes Muster liegt vor, das ausgeführt wird
        run_another_step = self.operations[self.next_op_index].step()
        if run_another_step:
            return True
        self.next_op_index += 1
        self.num_operations += 1

    if self.next_op_index == len(self.operations):
        # -> Einmal durch alle Operationen des Musters durchgelaufen
        # -> i zurücksetzen
        self.next_op_index = 0
        if self.check_for_changes:
            # Überprüfen, ob noch Veränderungen stattfinden
            new_sum = sum(
                [self.hof.felder[index] for index in self.source_fields])
            if new_sum == self.current_sum:
                return False
            self.current_sum = int(new_sum)

        if isinstance(self.operations[self.next_op_index], Muster):
            self.operations[self.next_op_index].reset()

    # Überprüfen, ob maximale Anzahl an Operationen erreicht wurde:
    if self.num_operations >= self.num_max_operations:
        return False

    if max(
        [self.hof.felder[index] for index in self.source_fields]
    ) <= self.tolerated_amount:
        return False

```

```

        return True

def run(self):
    """
    Ruft self.step() solange auf, bis die Abbruchbedingungen erfüllt sind
    """
    self.reset()
    while self.step():
        pass

```

Klasse Solver2, die Methoden zum Durchführen des Blasprozesses beinhaltet:

Auch hier wurden wieder einige Kommentare entfernt. In der Original-Python-Datei (aufgabe1_solver2.py) ist der vollständig kommentierte Code zu finden.

```

class Solver2:
    """
    Dient zum Erstellen, Speichern, Ausführen von Strategien bzw. generalisierten Ablaufplänen
    nach Verfahren 2 (siehe Dokumentation)
    """

    def __init__(self, hof, *, tolerated_amount, max_muster_operations=100,
choose_faster_path=False):
        self.hof = hof
        self.tolerated_amount = tolerated_amount
        if tolerated_amount <= 0:
            print("tolerated_amount muss größer als Null sein.")
            exit()
        self.strategy = []
        self.max_muster_operations = max_muster_operations
        self.running_op_index = 0
        # Bekommen später einen Wert zugewiesen:
        self.Q = None # Hier wird der Index von Feld Q als Tupel gespeichert werden
        self.num_rotations = 0
        self.choose_faster_path = choose_faster_path

    def add_operation(self, operation):
        """
        Fügt eine Blasoperation oder ein Muster zum Ablaufplan hinzu. Macht zuvor die
        Initialrotation rückgängig (beim späteren Ausführen wird der Hof in seiner ursprünglichen
        Ausrichtung betrachtet, die Rotation erfolgt nur, da dies die Generierung des generalisierten
        Ablaufplans / der Strategie erleichtert).
        """
        self.strategy.append(self.rotate_blasoperation(operation) if isinstance(operation, dict)
else operation)

    def rotate_field_index(self, field_index) -> tuple:
        """
        Ermittelt den ursprünglichen Index des Felds (den es vor der Initialrotation hatte), das
        derzeit am Index field_index ist
        """
        x_size, y_size = int(self.hof.x_size), int(self.hof.y_size)
        for i in range(self.num_rotations):
            field_index = (y_size-1-field_index[1], field_index[0]) # field_index um 90° im
Uhrzeigersinn rotieren
            helper = int(y_size) # Eine Rotation um 90° sorgt dafür, dass sich x-Größe (Breite)
                                und y-Größe (Höhe) des Hofs tauschen
            y_size = int(x_size)
            x_size = int(helper)
        return field_index

    def rotate_direction_vector(self, vector : tuple) -> tuple:
        """

```

```

    Ermittelt die ursprüngliche Richtung des Vektors vector (den er vor der Initialrotation
hatte)
    """
    for r in range(self.num_rotations):
        # Vektor um 90° im Uhrzeigersinn rotieren:
        if vector[0] == 0:
            vector = (-vector[1],0)
        elif vector[1] == 0:
            vector = (0,vector[0])
    return vector

def rotate_blasoperation(self, blasoperation) -> dict:
    return dict(
        feld0=self.rotate_field_index(blasoperation["feld0"]),
        blow_direction=self.rotate_direction_vector(blasoperation["blow_direction"])
    )

```

Funktionen, die Muster zum Transferieren von Blättern zurückgeben (gehören beide zur Solver2-Klasse):

```

def corner_to_edge(self, source_corner_field, target_field) -> Muster:
    """
    Returns:
        Muster: Ein Muster, das bei Anwendung Laub vom Eckfeld source_corner_field auf das
        Randfeld target_field bläst
    """
    if not self.hof.are_adjacent(source_corner_field, target_field):
        return
    if not (self.hof.is_corner(source_corner_field) and self.hof.is_edge(target_field)):
        return
    orthogonal_direction = (target_field[0]-source_corner_field[0], target_field[1]-
    source_corner_field[1])
    # Die Richtung, die orthogonal zur Blasrichtung ist (entspricht dem Vektor target_field-
    source_corner)
    # Beide zu orthogonal_direction orthogonale Richtungen werden als mögliche Blasrichtungen
    durchprobiert:
    anti_blow_direction = self.hof.get_orthogonal_direction(orthogonal_direction)
    # Gegenrichtung / Gegenvektor zur Blasrichtung
    feld0 = (source_corner_field[0]+anti_blow_direction[0],
    source_corner_field[1]+anti_blow_direction[1]) # Erstes mögliches Startfeld
    blow_direction = (-anti_blow_direction[0], -anti_blow_direction[1])
    if not self.hof.does_exist(feld0): # Wenn das erste mögliche Startfeld nicht existiert,
        dann muss das andere mögliche Startfeld das Startfeld sein
        anti_blow_direction = blow_direction
        feld0 = (source_corner_field[0]+anti_blow_direction[0],
        source_corner_field[1]+anti_blow_direction[1])
        blow_direction = (-anti_blow_direction[0], -anti_blow_direction[1])
    muster = Muster(self, [source_corner_field], [dict(feld0=feld0,
        blow_direction=blow_direction)], self.tolerated_amount)
    return muster

def edge_to_mid(self, source_edge_field, target_field) -> Muster:
    """
    Returns:
        Muster: Ein Muster, das bei Anwendung Laub vom Randfeld source_edge_field und seinen
        beiden auf dem Rand liegenden Nachbarn auf das Nicht-Rand-oder-Eckfeld
        target_field bläst
    """
    if not self.hof.are_adjacent(source_edge_field, target_field):
        return
    if (not self.hof.is_edge(source_edge_field)) or self.hof.is_edge(target_field):
        return

```

```

orthogonal_direction = (source_edge_field[0]-target_field[0], source_edge_field[1]-
                        target_field[1]) # Die Richtung, die zum Rand hinzeigt
blow_direction = self.hof.get_orthogonal_direction(orthogonal_direction)
operations = []
operations.append(dict(feld0=target_field, blow_direction=orthogonal_direction)) # Den
Blasvorgang von der Mitte zur Kante hin ermitteln (dieser verteilt das Laub durch die
Seitenabtriebe auf den beiden Feldern neben source_edge_field)
# Die Blasvorgänge, die Laub durch Seitenabtriebe auf target_field transportieren,
# ermitteln
feld0_1 = (source_edge_field[0]+blow_direction[0]*2,
            source_edge_field[1]+blow_direction[1]*2)
blow_direction = (-blow_direction[0], -blow_direction[1])
if self.hof.does_exist(feld0_1):
    operations.append(dict(feld0=feld0_1, blow_direction=blow_direction))
else:
    operations.insert(0, dict(feld0=(source_edge_field[0]-blow_direction[0]-
orthogonal_direction[0], source_edge_field[1]-blow_direction[1]-orthogonal_direction[1]),
blow_direction=orthogonal_direction))
    feld0_2 = (source_edge_field[0]+blow_direction[0]*2,
                source_edge_field[1]+blow_direction[1]*2)
    blow_direction = (-blow_direction[0], -blow_direction[1])
    if self.hof.does_exist(feld0_2):
        operations.append(dict(feld0=feld0_2, blow_direction=blow_direction))
    else:
        operations.insert(0, dict(feld0=(source_edge_field[0]-blow_direction[0]-
orthogonal_direction[0], source_edge_field[1]-blow_direction[1]-orthogonal_direction[1]),
blow_direction=orthogonal_direction))
        if not (self.hof.does_exist(feld0_1) or self.hof.does_exist(feld0_2)):
            return # Beide möglichen Startfelder existieren nicht -> Hof hat Kantenlänge 3 ->
Kante kann nicht gecleared werden
source_fields = [source_edge_field, (source_edge_field[0]+blow_direction[0],
source_edge_field[1]+blow_direction[1]), (source_edge_field[0]-blow_direction[0],
source_edge_field[1]-blow_direction[1])]
muster = Muster(self, source_fields, operations, self.tolerated_amount)
return muster

```

Die Funktionen, die die in den Phasen des generalisierten Ablaufplans auszuführenden Muster und Blasoperationen festlegen (gehören alle zur Solver2-Klasse):

```

def clear_bottom_line(self):
    """
    1. Phase des generalisierten Ablaufs: Befreien der untersten Reihe des Hofs. Fügt die
    hierfür notwendigen Operationen zu self.strategy hinzu.
    """
    for start_x in range(self.hof.x_size-1, 0, -1):
        self.add_operation(dict(feld0=(start_x, self.hof.y_size-1), blow_direction=(-1, 0)))
        # Nicht-Eckfelder am unteren Rand leeren
    self.add_operation(self.corner_to_edge((self.hof.x_size-1, self.hof.y_size-
        1), (self.hof.x_size-1, self.hof.y_size-2))) # Ecke rechts unten leeren
    self.add_operation(self.corner_to_edge((0, self.hof.y_size-1), (0, self.hof.y_size-2)))
        # Ecke links unten leeren

def move_to_top_line(self):
    """
    2. Phase des generalisierten Ablaufs: Bläst das gesamte Laub in die oberste Reihe. Fügt
    die hierfür notwendigen Operationen zu self.strategy hinzu.
    """
    for start_y in range(self.hof.y_size-1, 1, -1):
        for start_x in range(self.hof.x_size-1, -1, -1):
            if not (start_x == self.Q[0] and (start_y <= self.Q[1]+1 or self.hof.y_size == 5)
and (self.Q[1] != 1)): # Feld Q und Felder über Q nicht unnötigerweise leeren, da hier nachher
das Laub sowieder wieder hintransportiert werden muss
                self.add_operation(dict(feld0=(start_x, start_y), blow_direction=(0, -1)))

```

```

def concentrate_top_line(self):
    """
        3. Phase des generalisierten Ablaufs: Konzentriert das Laub der obersten Reihe auf dem
        Randfeld, das sich in der selben Spalte wie Q befindet. Fügt die hierfür notwendigen
        Operationen zu self.strategy hinzu.
    """
    Q = self.Q
    source_fields_for_edgeclear = [(Q[0]-1,0), (Q[0],0), (Q[0]+1,0)]
    if not (0,0) in source_fields_for_edgeclear:
        self.add_operation(self.corner_to_edge((0,0),(1,0)))
        # Ecke links oben clearen, wenn sie in Phase 4 nicht Source-Feld wird
    if not (self.hof.x_size-1,0) in source_fields_for_edgeclear:
        self.add_operation(self.corner_to_edge((self.hof.x_size-1,0),(self.hof.x_size-2,0)))
        # Ecke rechts oben clearen, wenn sie in Phase 4 nicht Source-Feld wird
    # Nicht-Eckfelder der obersten Reihe auf Source-Felder konzentrieren:
    for start_x in range(0,self.hof.x_size-1):
        if (start_x+1,0) in source_fields_for_edgeclear:
            break
        else:
            operationen = [dict(feld0=(start_x,0), blow_direction=(1,0)),
dict(feld0=(start_x+2,2), blow_direction=(0,-1))]
            muster = Muster(self, [(start_x+1,0), (start_x+2,1)], operationen,
self.tolerated_amount) # Muster zum Leeren des Eckfelds (start_x+1,0) und zum gleichzeitigen
Beseitigen des entstehenden Seitenabtriebs auf Feld (start_x+2,0)
            self.add_operation(muster)
    for start_x in range(self.hof.x_size-1,0,-1):
        if (start_x-1,0) in source_fields_for_edgeclear:
            break
        else:
            operationen = [dict(feld0=(start_x,0), blow_direction=(-1,0)),
dict(feld0=(start_x-2,2), blow_direction=(0,-1))]
            muster = Muster(self, [(start_x-1,0), (start_x-2,1)], operationen,
self.tolerated_amount)
            self.add_operation(muster)

def transfer_to_Q(self):
    """
        4. Phase des generalisierten Ablaufs: Verschiebt das gesamte Laub auf Feld Q. Fügt die
        hierfür notwendigen Operationen zu self.strategy hinzu.
    """
    Q = self.Q
    self.add_operation(self.edge_to_mid((Q[0],0), (Q[0],1))) # Muster hinzufügen, dass Laub
    vom Eckfeld (Q[0],0) auf das Feld (Q[0],1) bläst

    if Q != (Q[0],1): # Wenn (Q[0],1) == Q, dann befindet sich das Laub jetzt bereits auf Q -
        # wenn nicht, müssen weitere Schritte ausgeführt werden
        self.add_operation(dict(feld0=(Q[0],0), blow_direction=(0,1))) # Das Laub wird vom
        # Rand aus auf (Q[0],2) geblasen - dabei entstehen aber Seitenabtriebe. Mit
        # diesen wird im Folgenden umgegangen.
        # Mit den in der vorherigen Blasoperation entstandenen Seitenabtrieben umgehen: Wenn
        möglich wird das Laub verlustfrei durch den Abtrieb bei B vorne auf das Feld (Q[0],2) geblasen.
        Wenn nicht, dann wird das im vorherigen Schritt abgetriebene Laub auf den Felder (Q[0]-1,3),
        (Q[0],3) und (Q[0]+1,3) versammelt.
        cleared_fields = []
        if self.hof.does_exist((Q[0]+3,2)):
            muster = Muster(self, [(Q[0]+1,2)], [dict(feld0=(Q[0]+3,2),blow_direction=(-
                1,0))], self.tolerated_amount)
            self.add_operation(muster)
            cleared_fields.append((Q[0]+1,2))
        elif not (self.hof.x_size == 5 and self.hof.y_size == 5 and Q == (2,2)): # In diesem
            Fall wird anders vorgegangen (siehe Code unten)

```

```

muster = Muster(self, [(Q[0]+1,2)],
                 [dict(feld0=(Q[0]+1,0),blow_direction=(0,1))], self.tolerated_amount)
self.add_operation(muster)
if self.hof.does_exist((Q[0]-3,2)):
    muster = Muster(self, [(Q[0]-1,2)], [dict(feld0=(Q[0]-
            3,2),blow_direction=(1,0))], self.tolerated_amount)
    self.add_operation(muster)
    cleared_fields.append((Q[0]-1,2))
elif not (self.hof.x_size == 5 and self.hof.y_size == 5 and Q == (2,2)):
    # In diesem Fall wird anders vorgegangen (siehe Code unten)
    muster = Muster(self, [(Q[0]-1,2)], [dict(feld0=(Q[0]-
            1,0),blow_direction=(0,1))], self.tolerated_amount)
    self.add_operation(muster)
if self.hof.does_exist((Q[0],5)):
    if Q != (Q[0],3):
        muster = Muster(self, [(Q[0],3)], [dict(feld0=(Q[0],5),blow_direction=(0,-
            1))], self.tolerated_amount)
        self.add_operation(muster)
        cleared_fields.append((Q[0],3))
    elif Q == (Q[0],3):
        cleared_fields.append((Q[0],3))
    if len(cleared_fields) != 3:
        # Wenn dies nicht möglich war, dann wird stattdessen ein komplexeres System aus
        # Mustern und Blasoperationen verwendet, um das Laub auf Feld Q zu befärdern
        if (self.hof.does_exist((Q[0],6)) or ((Q[0], 3) == Q and
self.hof.does_exist((Q[0],5))) and (self.hof.does_exist((Q[0]-3, 3)) or
self.hof.does_exist((Q[0]+3, 3))):
            # In diesem Sonderfall sind unter (Q[0],2) noch 4 weitere Felder, was ein
            # verlustfreies Blasen des unteren Seitenabtriebs auf Feld (Q[0],2)
            # ermöglicht
            if self.hof.does_exist((Q[0]-3, 3)):
                self.add_operation(dict(feld0=(Q[0]+2,3), blow_direction=(-1,0)))
                self.add_operation(Muster(self, [(Q[0]-1, 3)], [dict(feld0=(Q[0]-3,3),
                    blow_direction=(1,0))], self.tolerated_amount))
            else:
                self.add_operation(dict(feld0=(Q[0]-2,3), blow_direction=(1,0)))
                self.add_operation(Muster(self, [(Q[0]+1, 3)], [dict(feld0=(Q[0]+3,3),
                    blow_direction=(-1,0))], self.tolerated_amount))
            self.add_operation(Muster(self, [(Q[0], 4)], [dict(feld0=(Q[0],6),
                blow_direction=(0,-1))], self.tolerated_amount))
            if not Q != (Q[0], 3):
                # Das Laub befindet sich nach den vorherigen Schritten auf Feld (Q[0], 3)
                # - Wenn (Q[0], 3) == 3 dann müssen folglich keine weiteren Operationen
                # mehr durchgeführt werden
                self.add_operation(Muster(self, [(Q[0], 3)], [dict(feld0=(Q[0],5),
                    blow_direction=(0,-1))], self.tolerated_amount))
        elif self.hof.does_exist((Q[0],5)):
            # Wenn unter Feld (Q[0], 2) drei Felder frei sind, dann wird mit folgendem
            # Muster kontinuierlich Laub auf Feld (Q[0], 2 geblasen)
            operations = []
            operations.append(dict(feld0=(Q[0]-2,3), blow_direction=(1,0)))
            operations.append(dict(feld0=(Q[0]+2,3), blow_direction=(-1,0)))
            operations.append(dict(feld0=(Q[0],5), blow_direction=(0,-1)))
            source_fields = [(Q[0]-1,3), (Q[0],3), (Q[0]+1,3), (Q[0],4)]
            self.add_operation(Muster(self, source_fields, operations,
                self.tolerated_amount))
    elif self.hof.x_size == 5 and self.hof.y_size == 5 and Q == (2,2):
        # Im übrigbleibenden Fall sind unter Feld (Q[0], 2) nur zwei Felder frei, es
        # muss also damit umgegangen werden, dass das Feld (Q[0], 4) ein Randfeld ist
        # Da der Hof so rotiert wird, dass die lange Seite die y-Dimension ist, tritt
        # dieser Fall nur für Höfe mit den Maßen (5,5) und einem sich in der Hofmitte
        # befindenden Feld Q auf
        self.add_operation(dict(feld0=(0,2), blow_direction=(1,0)))
        if self.choose_faster_path:

```

```

        self.add_operation(dict(feld0=(2,4), blow_direction=(0,-1)))
        self.num_rotations -= 1
    self.add_operation(Muster(self, [(1,2)], [dict(feld0=(1,4),
        blow_direction=(0,-1))], self.tolerated_amount))
    self.add_operation(Muster(self, [(3,2)], [dict(feld0=(3,4),
        blow_direction=(0,-1))], self.tolerated_amount))
operations = []
operations.append(Muster(self, [(1,1),(2,1),(3,1)], [
    dict(feld0=(0,1), blow_direction=(1,0)),
    dict(feld0=(4,1), blow_direction=(-1,0)),
], self.tolerated_amount)
)
operations.append(self.edge_to_mid((2,0), (2,1)))
self.add_operation(Muster(self, [(1,1), (2,1), (3,1), (2,0)], operations,
    self.tolerated_amount))
if Q != (Q[0], 2):
    # Das Laub befindet sich nach Durchführen der vorherigen Schritte auf (Q[0], 2).
    # Wenn Q[1] > 2, dann muss das Laub noch nach unten geblasen werden. Da über
    # (Q[0], 2) zwei Felder sind, ist das nun verlustfrei durch den Abtrieb bei B
    # vorne problemlos möglich
    for y_start in range(0,Q[1]-2):
        self.add_operation(Muster(self, [(Q[0], y_start+2)], [dict(feld0=(Q[0],
            y_start), blow_direction=(0,1))], self.tolerated_amount))

```

Die Funktion, die die Erstellung des generalisierten Ablaufplans initiiert und die Initialrotation durchführt (gehört zur Solver2-Klasse):

```

def build_strategy(self, *, Q):
    """
    Entwirft eine Strategie bzw. einen generalisierten Ablaufplan
    """
    self.strategy = []
    if not self.hof.does_exist(Q):
        print("Q existiert nicht.")
        exit()
    if self.hof.is_edge(Q):
        print("Q darf kein Rand-/Eckfeld sein.")
        exit()
    if self.hof.x_size == 3 and self.hof.y_size == 3:
        return # Für Höfe, bei denen eine Dimension kleiner 3 ist, gibt es keine Lösung. Für
    Höfe der Dimensionen (3,3) ist es am besten, gar nichts zu machen

    # Hof rotieren, sodass sich Feld Q im Bereich oben links befindet (Initialrotation)
    num_rotations = 0
    for i in range(3):
        if self.hof.y_size == 3 and self.hof.x_size != 3:
            break
        if not self.hof.x_size == 3:
            if Q[1] <= math.ceil(self.hof.y_size/2)-1:
                if Q[0] == 1:
                    break
                if not ((Q[0] == 1 and Q[1] != 1) or (Q[0] == self.hof.x_size-2 and Q[1] !=
                    1)):
                    if self.hof.x_size <= self.hof.y_size:
                        break
    self.hof.felder = np.rot90(self.hof.felder, k=1, axes=(0, 1))
    self.hof.x_size = self.hof.felder.shape[0]
    self.hof.y_size = self.hof.felder.shape[1]
    Q = (Q[1], self.hof.y_size-1-Q[0])
    num_rotations += 1
    self.num_rotations = num_rotations
    self.Q = Q

```

```

# Strategie speichern
self.strategy.append("\n[Phase 1: Unterste Reihe entlauben]")
self.clear_bottom_line()
self.strategy.append("\n[Phase 2: Laub auf oberste Reihe blasen]")
self.move_to_top_line()
self.strategy.append("\n[Phase 3: Laub auf oberster Reihe konzentrieren]")
self.concentrate_top_line()
self.strategy.append("\n[Phase 4: Laub nach Feld Q transferieren]")
self.transfer_to_Q()

# Initialrotation rückgängig machen
for i in range(self.num_rotations):
    self.hof.felder = np.rot90(self.hof.felder, k=-1, axes=(0, 1))
    Q = (self.hof.y_size-1-Q[1], Q[0])
    self.hof.x_size = self.hof.felder.shape[0]
    self.hof.y_size = self.hof.felder.shape[1]
    self.Q = Q

def step(self, *, render_zwischenschritte=False):
    """
    Führt basierend auf self.strategy die nächste Blasoperation aus
    """
    if self.running_op_index < len(self.strategy):
        operation = self.strategy[self.running_op_index]
        if isinstance(operation, str):
            self.hof.blas_log.append(operation)
            if render_zwischenschritte:
                self.hof.render(title="Laubverteilung"
vor:"+operation+f"\n({self.hof.blae_counter} Blasoperationen ausgeführt)", Q=self.Q)
                self.running_op_index += 1
                operation = self.strategy[self.running_op_index]

        if isinstance(operation, dict):
            self.hof.blase(operation["feld0"], operation["blow_direction"])

        elif isinstance(operation, Muster):
            run_another_step = operation.step()
            if run_another_step:
                return

        self.running_op_index += 1
    
```

Grundaufgabe: Ansatz 3

Bei Ansatz 3 handelt es sich um eine Kombination von Ansatz 1 und 2. Daher kommen keine neuen Funktionen hinzu, die hier dokumentiert werden müssten.

Auch bei den vorgenommenen Erweiterungen (die alle auf Ansatz 2 basieren) werden nur die Funktionen, Klassen und Datenstrukturen dokumentiert, die sich verändern bzw. neu dazukommen.

Erweiterung 1 (basierend auf Ansatz 2)

Veränderte Funktionen, die die in den Phasen des generalisierten Ablaufplans auszuführenden Muster und Blasoperationen festlegen (Teil der Solver2-Klasse):

```

def clear_edges(self):
    """
    1. Phase: Ränder (bis auf oberen Rand) entlauben, um zu vermeiden, dass
    beim Leeren der Ecken Laub verloren geht
    """
    
```

```

for start_x in range(self.hof.x_size-1, 3, -1):
    self.add_operation(dict(feld0=(start_x, self.hof.y_size-
        1), blow_direction=(-1, 0))) # Nicht-Eckfelder am unteren Rand leeren
if self.hof.x_size > 4:
    self.add_operation(self.edge_to_mid((2, self.hof.y_size-1), (2,
        self.hof.y_size-2)))
elif self.hof.x_size > 3:
    self.add_operation(self.edge_to_mid((1, self.hof.y_size-1), (1,
        self.hof.y_size-2)))
self.add_operation(self.corner_to_edge((self.hof.x_size-1,
    self.hof.y_size-1), (self.hof.x_size-1, self.hof.y_size-2))) # Ecke
    rechts unten leeren
self.add_operation(self.corner_to_edge((0, self.hof.y_size-
    1), (0, self.hof.y_size-2))) # Ecke links unten leeren
for start_y in range(self.hof.y_size-1, 3, -1):
    self.add_operation(dict(feld0=(0, start_y), blow_direction=(0, -1)))
    # Nicht-Eckfelder am unteren Rand leeren
for start_y in range(self.hof.y_size-1, 3, -1):
    self.add_operation(dict(feld0=(self.hof.x_size-
        1, start_y), blow_direction=(0, -1))) # Nicht-Eckfelder am unteren
    Rand leeren
if self.hof.y_size > 4:
    self.add_operation(self.edge_to_mid((self.hof.x_size-1, 2),
        (self.hof.x_size-2, 2)))
    self.add_operation(self.edge_to_mid((0, 2), (1, 2)))
elif self.hof.y_size > 3:
    self.add_operation(self.edge_to_mid((self.hof.x_size-1, 1),
        (self.hof.x_size-2, 1)))
    self.add_operation(self.edge_to_mid((0, 1), (1, 1)))
for start_x in range(0, self.Q[0]-1):
    for start_y in range(self.hof.y_size-3, 0, -1):
        if start_x == self.Q[0]-2 and self.hof.does_exist((start_x-1,
            start_y)):
            self.add_operation(Muster(self, [(start_x+1, start_y)],
                [dict(feld0=(start_x-1, start_y), blow_direction=(1, 0))],
                self.tolerated_amount))
        else:
            self.add_operation(dict(feld0=(start_x, start_y),
                blow_direction=(1, 0)))
for start_x in range(self.hof.x_size-1, self.Q[0]+1, -1):
    for start_y in range(self.hof.y_size-3, 0, -1):
        if start_x == self.Q[0]+2 and self.hof.does_exist((start_x+1,
            start_y)):
            self.add_operation(Muster(self, [(start_x-1, start_y)],
                [dict(feld0=(start_x+1, start_y), blow_direction=(-1, 0))],
                self.tolerated_amount))
        else:
            self.add_operation(dict(feld0=(start_x, start_y),
                blow_direction=(-1, 0)))
if self.hof.does_exist((Q[0]-3, self.hof.y_size-2)):
    self.add_operation(Muster(self, [(Q[0]-1, self.hof.y_size-2)],
        [dict(feld0=(Q[0]-3, self.hof.y_size-2), blow_direction=(1, 0))],
        self.tolerated_amount))

```

```

        if self.hof.does_exist((Q[0]+3, self.hof.y_size-2)):
            self.add_operation(Muster(self, [(Q[0]+1, self.hof.y_size-2)],
            [dict(feld0=(Q[0]+3, self.hof.y_size-2), blow_direction=(-1,0))],
            self.tolerated_amount))

    def move_to_top_line(self):
        """
        2. Phase des generalisierten Ablaufs: Bläst das gesamte Laub in die
        oberste Reihe. Fügt die hierfür notwendigen Operationen zu self.strategy hinzu.
        """
        for start_y in range(self.hof.y_size-1, 1, -1):
            for start_x in range(self.hof.x_size-1, -1, -1):
                if not (start_x == self.Q[0] and (start_y <= self.Q[1]+1 or
                self.hof.y_size == 5)):
                    if not (start_x == self.Q[0] and start_y-1 == self.Q[1]):
                        self.add_operation(dict(feld0=(start_x, start_y),
                        blow_direction=(0, -1)))
                elif start_y > 3:
                    self.add_operation(Muster(
                        [start_x, start_y-1], [dict(feld0=(start_x,
                        start_y), blow_direction=(0, -1))],
                        self.tolerated_amount
                    ))
    def concentrate_top_line(self):
        """
        3. Phase des generalisierten Ablaufs: Konzentriert das Laub der
        obersten Reihe auf dem Randfeld, das sich in der selben Spalte wie Q befindet.
        Fügt die hierfür notwendigen Operationen zu self.strategy hinzu.
        """
        Q = self.Q
        source_fields_for_edgeclear = [(Q[0]-1, 0), (Q[0], 0), (Q[0]+1, 0)]
        if not (0, 0) in source_fields_for_edgeclear:
            self.add_operation(self.corner_to_edge((0, 0), (1, 0))) # Ecke links
        oben clearen, wenn sie in Phase 4 nicht Source-Feld wird
        if not (self.hof.x_size-1, 0) in source_fields_for_edgeclear:
            self.add_operation(self.corner_to_edge((self.hof.x_size-
            1, 0), (self.hof.x_size-2, 0))) # Ecke rechts oben clearen, wenn sie in Phase 4
        nicht Source-Feld wird
        # Nicht-Eckfelder der obersten Reihe auf Source-Felder konzentrieren:
        for start_x in range(0, self.hof.x_size-1):
            if (start_x+1, 0) in source_fields_for_edgeclear:
                break
            else:
                operationen = [dict(feld0=(start_x, 0), blow_direction=(1, 0)),
                dict(feld0=(start_x+2, 0), blow_direction=(0, -1))]
                muster = Muster(self, [(start_x+1, 0), (start_x+2, 0)],
                operationen, self.tolerated_amount) # Muster zum Leeren des Eckfelds
                (start_x+1, 0) und zum gleichzeitigen Beseitigen des entstehenden Seitenabtriebs
                auf Feld (start_x+2, 0)
                self.add_operation(muster)
        for start_x in range(self.hof.x_size-1, 0, -1):
            if (start_x-1, 0) in source_fields_for_edgeclear:

```

```

        break
    else:
        operationen = [dict(feld0=(start_x,0), blow_direction=(-1,0)),
dict(feld0=(start_x-2,2), blow_direction=(0,-1))]
            muster = Muster(self, [(start_x-1,0), (start_x-2,1)],
operationen, self.tolerated_amount)
            self.add_operation(muster)

def transfer_to_Q(self):
    """
    4. Phase des generalisierten Ablaufs: Verschiebt das gesamte Laub auf
    Feld Q. Fügt die hierfür notwendigen Operationen zu self.strategy hinzu.
    """

    Q = self.Q
    self.add_operation(self.edge_to_mid((Q[0],0), (Q[0],1))) # Muster
hinzufügen, dass Laub vom Eckfeld (Q[0],0) auf das Feld (Q[0],1) bläst
    if Q == (Q[0],1):
        if not self.hof.is_edge((Q[0],2)):
            if self.hof.does_exist((Q[0],4)):
                self.add_operation(Muster(self, [(Q[0],2)],
[dict(feld0=(Q[0],4), blow_direction=(0,-1))], self.tolerated_amount))
            else:
                self.add_operation(dict(feld0=(Q[0],3), blow_direction=(0,-
1)))
        else:
            self.add_operation(dict(feld0=(Q[0],0), blow_direction=(0,1))) #
Das Laub wird vom Rand aus auf (Q[0],2) geblasen - dabei entstehen aber
Seitenabtriebe. Mit diesen wird im Folgenden umgegangen.
            # Mit den in der vorherigen Blasoperation entstandenen
Seitenabtrieben umgehen: Wenn möglich wird das Laub verlustfrei durch den
Abtrieb bei B vorne auf das Feld (Q[0],2) geblasen. Wenn nicht, dann wird das
im vorherigen Schritt abgetriebene Laub auf den Felder (Q[0]-1,3), (Q[0],3) und
(Q[0]+1,3) versammelt.
            cleared_fields = []
            if self.hof.does_exist((Q[0]+3,2)):
                muster = Muster(self, [(Q[0]+1,2)],
[dict(feld0=(Q[0]+3,2),blow_direction=(-1,0))], self.tolerated_amount)
                self.add_operation(muster)
                cleared_fields.append((Q[0]+1,2))
            elif not (self.hof.x_size == 5 and self.hof.y_size == 5 and Q ==
(2,2)): # In diesem Fall wird anders vorgegangen (siehe Code unten)
                muster = Muster(self, [(Q[0]+1,2)],
[dict(feld0=(Q[0]+1,0),blow_direction=(0,1))], self.tolerated_amount)
                self.add_operation(muster)
                if self.hof.does_exist((Q[0]-3,2)):
                    muster = Muster(self, [(Q[0]-1,2)], [dict(feld0=(Q[0]-
3,2),blow_direction=(1,0))], self.tolerated_amount)
                    self.add_operation(muster)
                    cleared_fields.append((Q[0]-1,2))
                elif not (self.hof.x_size == 5 and self.hof.y_size == 5 and Q ==
(2,2)): # In diesem Fall wird anders vorgegangen (siehe Code unten)
                    muster = Muster(self, [(Q[0]-1,2)], [dict(feld0=(Q[0]-
1,0),blow_direction=(0,1))], self.tolerated_amount)

```

```

        self.add_operation(muster)
if self.hof.does_exist((Q[0],5)):
    if Q != (Q[0],3):
        muster = Muster(self, [(Q[0],3)],
                         [dict(feld0=(Q[0],5), blow_direction=(0,-1))],
                         self.tolerated_amount)
        self.add_operation(muster)
        cleared_fields.append((Q[0],3))
elif Q == (Q[0],3):
    cleared_fields.append((Q[0],3))
if len(cleared_fields) != 3:
    if (self.hof.does_exist((Q[0],6)) or ((Q[0], 3) == Q and
        self.hof.does_exist((Q[0],5)))) and
        (self.hof.does_exist((Q[0]-3, 3)) or
        self.hof.does_exist((Q[0]+3, 3))):
        if self.hof.does_exist((Q[0]-3, 3)):
            self.add_operation(dict(feld0=(Q[0]+2,3),
                                    blow_direction=(-1,0)))
            self.add_operation(Muster(self, [(Q[0]-1, 3)],
                                      [dict(feld0=(Q[0]-3,3), blow_direction=(1,0))],
                                      self.tolerated_amount))
        else:
            self.add_operation(dict(feld0=(Q[0]-2,3),
                                    blow_direction=(1,0)))
            self.add_operation(Muster(self, [(Q[0]+1, 3)],
                                      [dict(feld0=(Q[0]+3,3), blow_direction=(-1,0))],
                                      self.tolerated_amount))
            self.add_operation(Muster(self, [(Q[0], 4)],
                                      [dict(feld0=(Q[0],6), blow_direction=(0,-1))],
                                      self.tolerated_amount))
    if not Q != (Q[0], 3):
        self.add_operation(Muster(self, [(Q[0], 3)],
                                  [dict(feld0=(Q[0],5), blow_direction=(0,-1))],
                                  self.tolerated_amount))
elif self.hof.does_exist((Q[0],5)):
    operations = []
    operations.append(dict(feld0=(Q[0]-2,3),
                           blow_direction=(1,0)))
    operations.append(dict(feld0=(Q[0]+2,3), blow_direction=(-1,0)))
    operations.append(dict(feld0=(Q[0],5), blow_direction=(0,-1)))
    source_fields = [(Q[0]-1,3), (Q[0],3), (Q[0]+1,3),
                      (Q[0],4)]
    self.add_operation(Muster(self, source_fields, operations,
                               self.tolerated_amount))
    elif self.hof.x_size == 5 and self.hof.y_size == 5 and Q ==
(2,2):
        self.add_operation(dict(feld0=(0,2), blow_direction=(1,0)))
        if self.choose_faster_path:
            self.add_operation(dict(feld0=(2,4),
                                   blow_direction=(0,-1)))
            self.num_rotations += 1

```

```

        self.add_operation(Muster(self, [(1,2)], [dict(feld0=(1,4),
blow_direction=(0,-1))]), self.tolerated_amount)
        self.add_operation(Muster(self, [(3,2)], [dict(feld0=(3,4),
blow_direction=(0,-1))]), self.tolerated_amount)
    operations = []
    operations.append(Muster(self, [(1,1),(2,1),(3,1)], [
        dict(feld0=(0,1), blow_direction=(1,0)),
        dict(feld0=(4,1), blow_direction=(-1,0)),
    ], self.tolerated_amount))
)
operations.append(self.edge_to_mid((2,0), (2,1)))
self.add_operation(Muster(self, [(1,1), (2,1), (3,1),
(2,0)], operations, self.tolerated_amount))
if Q != (Q[0], 2):
    for y_start in range(0,Q[1]-2):
        self.add_operation(Muster(self, [(Q[0], y_start+2)],
[dict(feld0=(Q[0], y_start), blow_direction=(0,1))], self.tolerated_amount))

```

Erweiterung 2 (basierend auf Ansatz 2)

Veränderte Funktionen zum Laubblasen, die jetzt zwei Laubbläser gleichzeitig behandeln (Teil der Hof-Klasse):

```

def run_single_blasoperation(self, feld0 : tuple, blow_direction : tuple,
blocked_fields : list[tuple], *, remaining_single_ops=0):
    """
        Führt die Blasoperation eines einzigen Laubbläasers aus und aktualisiert
        self.felder entsprechend.

        Args:
            feld0 (tuple), blow_direction (tuple): Index von Feld 0 und
            Blasrichtung
            blocked_fields (list[tuple]): Felder, auf die kein Laub gelangen
            darf (da sie im Einflussgebiets eines anderen Laubbläasers liegen) und die daher
            wie Randfelder behandelt werden
    """
    if not blow_direction in [(0,1),(0,-1),(1,0),(-1,0)]:
        return # -> Ungültige Blasrichtung. Der Laubbläser kann nur nach
rechts, links, oben und unten blasen.

    # Richtung, die orthogonal zur Blasrichtung ist, ermitteln:
    orthogonal_direction = self.get_orthogonal_direction(blow_direction)
    # Feld A (Feld unmittelbar vor dem Laubbläser) ermitteln:
    feldA = (feld0[0]+blow_direction[0], feld0[1]+blow_direction[1])
    if not self.does_exist(feldA):
        return # -> Es gibt kein Feld vor dem Laubbläser bzw. der
Laubbläser bläst gegen die Umrandung. Für diesen Fall ist definiert, dass sich
die Verteilung des Laubs nicht verändert
    new_feldA_value = 0 # In dieser Variable wird die neue Anzahl an
Blättern auf Feld A gespeichert

    # Feld B (Feld hinter Feld A) ermitteln:

```

```

feldB = (feldA[0]+blow_direction[0], feldA[1]+blow_direction[1])
if self.does_exist(feldB) and feldB not in blocked_fields:
    # -> Es gibt ein Feld B. Neue Anzahl an Blättern auf Feld B
ermitteln:
    if self.rules.use_binomial:
        # -> Zum Modellieren der Blätteranzahlen sollen ganze Zahlen
verwendet werden, die über die Binomialverteilung bestimmt werden
        A_seitenabtrieb_1, _ = binomial_likeiest(n=self.felder[feldA],
p=self.rules.A_seitenabtrieb, rank=self.rules.binomial_rank,
handle_ties=self.rules.binomial_handle_ties) # Anzahl an Blättern von Feld A,
die auf dem einen Feld neben Feld B landen (Seitenabtrieb 1)
        A_seitenabtrieb_2, _ = binomial_likeiest(n=self.felder[feldA]-A_seitenabtrieb_1, p=self.rules.A_seitenabtrieb/(1-self.rules.A_seitenabtrieb),
rank=self.rules.binomial_rank, handle_ties=self.rules.binomial_handle_ties) # Anzahl an Blättern von Feld A, die auf dem anderen Feld neben Feld B landen
(Seitenabtrieb 2)
        B_vorne_abtrieb, _ = binomial_likeiest(n=self.felder[feldB],
p=self.rules.B_vorne_abtrieb, rank=self.rules.binomial_rank,
handle_ties=self.rules.binomial_handle_ties) # Anzahl an Blättern von Feld B,
die nach vorne abgetrieben werden
    else:
        # -> Zum Modellieren der Blätteranzahlen werden die
Erwartungswerte (als Fließkommazahlen) verwendet
        A_seitenabtrieb_1 = self.felder[feldA] *
self.rules.A_seitenabtrieb
        A_seitenabtrieb_2 = A_seitenabtrieb_1
        B_vorne_abtrieb = self.felder[feldB] *
self.rules.B_vorne_abtrieb
        new_feldB_value = self.felder[feldA] - (A_seitenabtrieb_1 +
A_seitenabtrieb_2) + self.felder[feldB] - B_vorne_abtrieb # Neue Anzahl an
Blättern auf Feld B

        # Nachbarfeld von Feld B, das Feld A gegenüberliegt, aktualisieren
(sofern vorhanden):
        n1 = (feldB[0]+blow_direction[0], feldB[1]+blow_direction[1]) #
Index des Nachbarfelds
        if self.does_exist(n1) and n1 not in blocked_fields:
            self.felder[n1] += B_vorne_abtrieb
        else:
            new_feldB_value += B_vorne_abtrieb

        # Nachbarfelder von Feld B, die Feld A nicht gegenüberliegen,
aktualisieren (sofern vorhanden):
        n1 = (feldB[0]+orthogonal_direction[0],
feldB[1]+orthogonal_direction[1])
        if self.does_exist(n1) and n1 not in blocked_fields:
            self.felder[n1] += A_seitenabtrieb_1
        else:
            new_feldB_value += A_seitenabtrieb_1
        n2 = (feldB[0]-orthogonal_direction[0], feldB[1]-
orthogonal_direction[1])
        if self.does_exist(n2) and n2 not in blocked_fields:
            self.felder[n2] += A_seitenabtrieb_2

```

```

        else:
            new_feldB_value += A_seitenabtrieb_2

            # Feld B aktualisieren
            self.felder[feldB] = new_feldB_value
    else:
        # -> Es gibt kein Feld B.
        if self.rules.use_binomial:
            A_noB_seitenabtrieb_1, _ =
binomial_likeyest(n=self.felder[feldA], p=self.rules.A_noB_seitenabtrieb,
rank=self.rules.binomial_rank, handle_ties=self.rules.binomial_handle_ties)
            A_noB_seitenabtrieb_2, _ =
binomial_likeyest(n=self.felder[feldA]-A_noB_seitenabtrieb_1,
p=self.rules.A_noB_seitenabtrieb/(1-self.rules.A_noB_seitenabtrieb),
rank=self.rules.binomial_rank, handle_ties=self.rules.binomial_handle_ties)
            else:
                A_noB_seitenabtrieb_1 = self.felder[feldA] *
self.rules.A_noB_seitenabtrieb
                A_noB_seitenabtrieb_2 = A_noB_seitenabtrieb_1
                new_feldA_value = self.felder[feldA] - (A_noB_seitenabtrieb_1 +
A_noB_seitenabtrieb_2)

                # Nachbarfelder von Feld A, die dem laubblasenden Hausmeister nicht
gegenüberliegen, aktualisieren, sofern vorhanden:
                n1 = (feldA[0]+orthogonal_direction[0],
feldA[1]+orthogonal_direction[1])
                if self.does_exist(n1) and n1 not in blocked_fields:
                    self.felder[n1] += A_noB_seitenabtrieb_1
                else:
                    # -> Feld A ist ein Eckfeld, ein Teil des Laubs verbleibt also
auf Feld A.
                    new_feldA_value += A_noB_seitenabtrieb_1
                    n2 = (feldA[0]-orthogonal_direction[0], feldA[1]-
orthogonal_direction[1])
                    if self.does_exist(n2) and n2 not in blocked_fields:
                        self.felder[n2] += A_noB_seitenabtrieb_2
                    else:
                        # -> Feld A ist ein Eckfeld, ein Teil des Laubs verbleibt also
auf Feld A.
                        new_feldA_value += A_noB_seitenabtrieb_2

                # Neuen Wert von A zurückgeben (Aktualisierung des Feldwerts erfolgt am
Ende des insgesamten Blasvorgangs, damit die verschiedenen Blasvorgänge alle
vom selben Ausgangspunkt ausgehen):
                return new_feldA_value

```

```
def blase(self, feld0_b1, blow_direction_b1, feld0_b2, blow_direction_b2):
    """

```

Simuliert einen Blasvorgang (mit zwei Laubbläsern) und aktualisiert self.felder auf die resultierende Blattverteilung.

```

Args:
    feld0_b1 (int): Index des Felds, auf dem der Hausmeister 1 mit
Bläser 1 steht
        blow_direction_b1 (tuple): Richtung, in die der Hausmeister 1 mit
Bläser 1 bläst. Kann folgende Werte annehmen: (1,0) (=rechts), (-1,0) (=links),
(0,1) (=unten), (0,-1) (=oben)
    feld0_b2 (int): Index des Felds, auf dem der Hausmeister 2 mit
Bläser 2 steht
        blow_direction_b2 (tuple): Richtung, in die der Hausmeister 2 mit
Bläser 2 bläst. Kann folgende Werte annehmen: (1,0) =rechts, (-1,0) (=links),
(0,1) (=unten), (0,-1) (=oben)
    """
    if feld0_b2 is None:
        feldA_b1 = (feld0_b1[0]+blow_direction_b1[0],
feld0_b1[1]+blow_direction_b1[1])
            self.felder[feldA_b1] = self.run_single_blasoperation(feld0_b1,
blow_direction_b1, [])
            return
    if feld0_b1 == feld0_b2:
        return None # Laubbläser dürfen nicht auf selbem Feld stehen
    pd = [(0,1),(0,-1),(1,0),(-1,0)] # Mögliche Blasrichtungen
    if not blow_direction_b1 in pd and blow_direction_b2 in pd:
        return # -> Ungültige Blasrichtung. Der Laubbläser kann nur nach
rechts, links, oben und unten blasen.
        self.blas_counter += 1 # Zähler, der durchgeführte Blasoperationen
zählt, erhöhen
        self.blas_log.append(dict(feld0_b1=feld0_b1,
blow_direction_b1=blow_direction_b1, feld0_b2=feld0_b2,
blow_direction_b2=blow_direction_b2)) # Blasoperation loggen

        feldA_b1 = (feld0_b1[0]+blow_direction_b1[0],
feld0_b1[1]+blow_direction_b1[1])
        if not self.does_exist(feldA_b1): # Überprüfen, ob das ermittelte Feld
tatsächlich existiert
            feldA_b1 = None
        feldA_b2 = (feld0_b2[0]+blow_direction_b2[0],
feld0_b2[1]+blow_direction_b2[1])
        if not self.does_exist(feldA_b2):
            feldA_b2 = None
        if feldA_b1 == feld0_b2 and feld0_b1 == feldA_b2:
            return

        # Blasoperation für den ersten Laubbläser (b1) ausführen:
        if feldA_b2 == feldA_b1:
            half_feldA_value = round(self.felder[feldA_b2]/2)
            self.felder[feldA_b2] -= half_feldA_value
            blocked_fields = [feldA_b2]
            if manhattan_distance(feld0_b2, feld0_b1) <
manhattan_distance(feld0_b2, (feld0_b1[0]+blow_direction_b2[0],
feld0_b1[1]+blow_direction_b2[1])):
                blocked_fields.append(feld0_b2)
            new_feldA_b1_value = self.run_single_blasoperation(feld0_b1,
blow_direction_b1, blocked_fields)

```

```
if feldA_b2 == feldA_b1:  
    self.felder[feldA_b2] = half_feldA_value  
blocked_fields = [feldA_b1]  
    if manhattan_distance(feld0_b1, feld0_b2) <  
manhattan_distance(feld0_b1, (feld0_b2[0]+blow_direction_b1[0],  
feld0_b2[1]+blow_direction_b1[1])):  
        blocked_fields.append(feld0_b1)  
    new_feldA_b2_value = self.run_single_blasoperation(feld0_b2,  
blow_direction_b2, blocked_fields)  
  
if feldA_b2 == feldA_b1:  
    self.felder[feldA_b1] = new_feldA_b1_value + new_feldA_b2_value  
else:  
    if not new_feldA_b1_value is None:  
        self.felder[feldA_b1] = new_feldA_b1_value  
    if not new_feldA_b2_value is None:  
        self.felder[feldA_b2] = new_feldA_b2_value
```