



Alles Käse

? A2  64712  Leonhard Masche  15.04.2023

Inhaltsverzeichnis

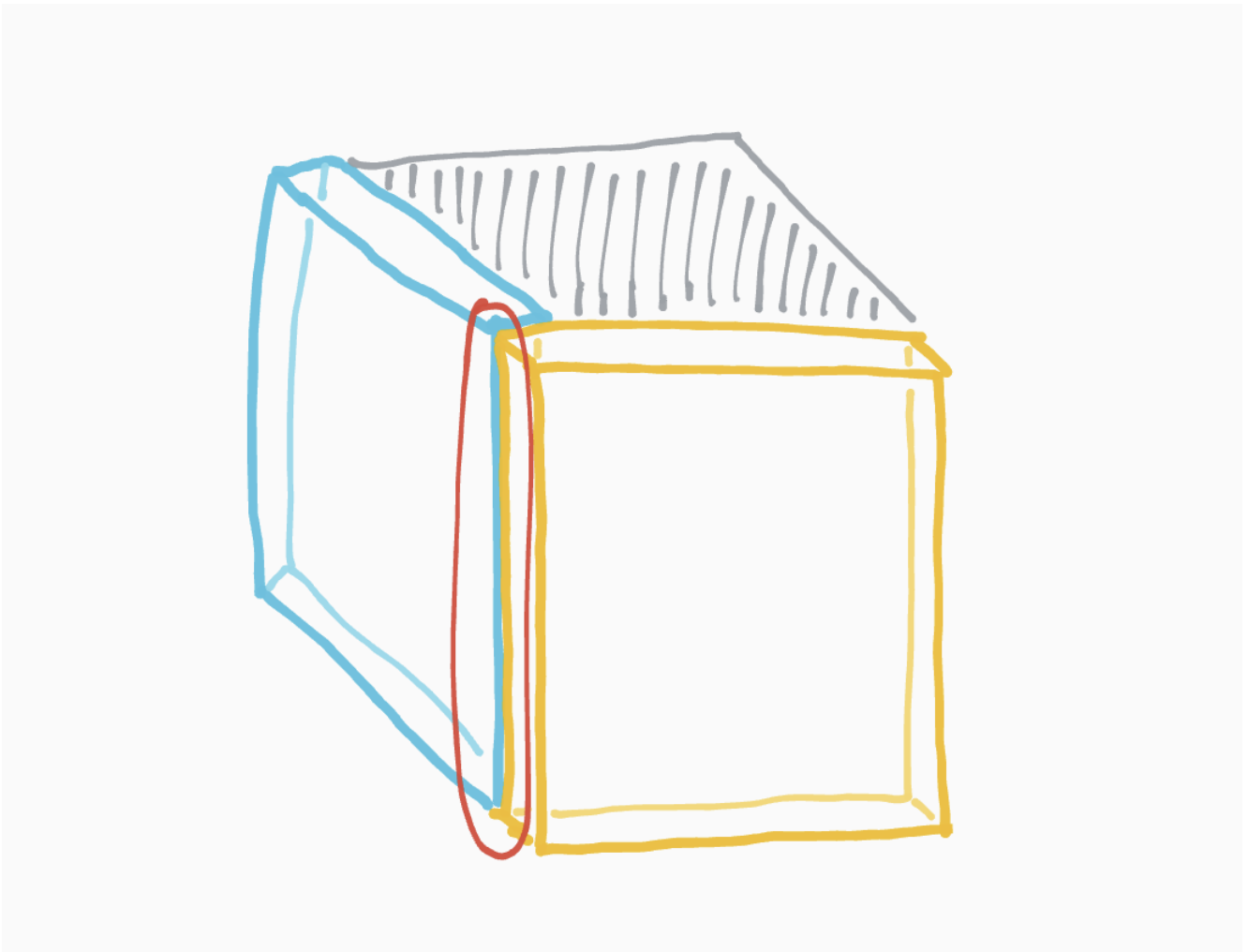
1. [Lösungsidee](#)
2. [Umsetzung](#)
 - i. [Verbesserungen](#)
 - ii. [Laufzeit](#)
 - iii. [Komplexität](#)
 - iv. [Einzigartigkeit](#)
3. [Beispiele](#)
4. [Quellcode](#)

Lösungsidee

Die Aufgabe wird als ein kompletter Graph $G(V, E)$ dargestellt. Hierbei sind die Knoten V die Scheiben, und die Kanten E repräsentieren ein Aufeinanderfolgen dieser Scheiben. Nun gilt es einen Hamiltonpfad in diesem Graphen zu finden, der die geometrischen Bedingungen der orthogonalen Schnitte erfüllt. Existiert dieser, gibt es für diese Käsescheiben eine Lösung und der Hamilton-Pfad (startend von der End-Scheibe mit der kleineren Fläche) ist die Reihenfolge, in der die Scheiben wieder zusammengefügt werden können.

Um zu sehen, ob zwei Scheiben überhaupt aufeinander folgen können, wird die folgende Beobachtung verwendet:

Lemma 1: Auf den Quader kann eine Scheibe nur nach einer anderen Scheibe hinzugefügt werden, wenn sie in mindestens einer der beiden Größen mit der vorherigen übereinstimmt.



Beweis: Wenn eine Scheibe nach einer anderen hinzugefügt wird, muss sie sich mit ihr mindestens eine Kante von gleicher Länge teilen.

Eine Scheibe, die diese Bedingungen erfüllt, passt aber nicht immer auf den Quader. Zusätzlich muss während des Aufbaus also noch geprüft werden, ob die Scheibe wirklich die gleichen Dimensionen wie eine Seite des aktuellen Quaders hat.

Durch einen Backtracking-Algorithmus werden mögliche Nachbarn ausprobiert, bis die Lösung gefunden wurde. Alle Kombinationen auszuprobieren scheint ineffizient, lässt sich aber durch ein paar Tricks so optimieren, dass auch unter **1.5** Millionen Scheiben noch eine Lösung gefunden werden kann.

Umsetzung

Zuerst werden die Scheiben in eine Liste geladen. Aus dieser wird nun eine Lookup-Tabelle von Seitenlänge zum Index in der Liste erstellt, um effizienter auf potentiell anfügbare Scheiben zugreifen zu können. So werden Pfade, die sicher nicht zu einem Ergebnis führen vorzeitig ausgeschlossen. Nun wird ein Backtracking-Algorithmus angewendet. Es werden immer weiter passende Scheiben hinzugefügt, und falls keine weitere Lösung möglich ist, wird der Pfad zurückverfolgt, bis es weitere mögliche Nachbarn gibt und dieser Pfad wird genauso weiterverfolgt. Wird die Lösung gefunden, wird sie zurückgegeben.

Das Programm (`program.py`) ist in Python geschrieben und mit einer Umgebung ab der Version `3.8` ausführbar. Es werden nur Standard-Bibliotheken verwendet. Wird das Programm aufgerufen, fragt es nach der Zahl des Beispiels und berechnet die Lösung für dieses anschließend. Zusätzlich wird diese Lösung für die BWINF-Beispiele in Textform in dem Ordner `output` gespeichert. Jede Zeile beschreibt eine Scheibe aus dem Beispiel in der Reihenfolge, in der sie hinzugefügt werden. Bonus: Ein Programm zur Verifizierung der Ergebnisse befindet sich im IPython-Notebook `test.ipynb` .

Verbesserungen

Deduplizierung

Nachbarn mit gleicher Größe werden dedupliziert. So wird ein unnötiges mehrfaches Besuchen dieser Nachbarn verhindert, welches garantiert nicht zu einer Lösung führt, da dieser Lösungsweg schon einmal besucht wurde.

Aufgegessen

Da hatte Antje doch zu viel Hunger und hat einige Scheiben aufgegessen! Eine modifizierte Version des Programmes kann auch Beispiele lösen, in denen Scheiben fehlen. Dazu werden nicht nur nur Nachbarn mit den passenden Dimensionen, sondern auch Nachbarn, die in jeweils einer Dimension um **1** größer sind überprüft. Tritt ein solcher Fall ein, wird dem Pfad eine 'virtuelle' Scheibe hinzugefügt, und weiter iteriert. So kann das Programm einzelne Scheiben die im Stapel fehlen wiederherstellen. Sollten zwei oder mehr Scheiben in Folge fehlen, werden die Scheiben auf mehrere Quader verteilt (siehe nächster Punkt). Da nun statt maximal drei Möglichkeiten, eine Scheibe anzufügen, **12** Möglichkeiten betrachtet werden, steigt der Rechenaufwand auch sehr schnell mit der Länge des Beispiels.

Mehr Käse

Auch wenn Antje an einem Tag mehrfach telefoniert, und alle Käsescheiben vermischt hat, kann auch dieses Problem gelöst werden. Dazu werden von jedem Startknoten aus alle Pfade mit maximaler Länge generiert. Diese müssen aber nicht vollständig sein. Nun wird jede Kombination aus ***n*** Pfaden überprüft, wobei ***n*** von **1** bis hin zur Länge des Käsestapels erhöht wird. Sind in einer Kombination zu viele Knoten enthalten, werden sie vom Ende der Pfade entfernt. Wurden Lösungen gefunden, wird die mit den wenigsten 'aufgegessenen' Scheiben zurückgegeben.

Laufzeit

Im Folgenden wird Die Laufzeit des ursprünglichen Algorithmus analysiert.

Bei solchen Problemen liegt es nahe, einfach alle Kombinationen auszuprobieren, was eine Laufzeit von $O(n!)$ bedeuten würde.

Nun kann man aber feststellen, dass an einen Quader von beliebiger Größe $a \times b \times c$ nur maximal drei Scheiben ($a \times b$, $b \times c$ und $a \times c$) angefügt werden können. Weitere Scheiben mit den gleichen Maßen können vernachlässigt werden, da diese logischerweise zu derselben Lösung führen würden. Wenn nun also für jede mögliche Start-Scheibe alle Kombinationen ausprobiert werden, ergibt sich eine Worst-Case Zeitkomplexität von $O(n * 3^{n-1})$, wobei die Basis **3** die maximale Anzahl der Nachbarn ist.

Das ist nun aber die Worst-Case Laufzeit des Programmes. In Wirklichkeit liegt die (experimentell ermittelte) durchschnittliche Anzahl an Nachbarn während dem Lösungsvorgang zwischen **1.00000** und **1.04167**.

Somit befindet sich auch die Zeitkomplexität im Bereich zwischen $O(n1.00^{n-1}) \approx O(n)$ und $O(n1.04^{n-1})$.

Komplexität

Das Problem (eine Lösung zu finden) kann auf ein Hamiltonian-Path-Problem reduziert werden. Dieses kann wiederum über z. B. Integer Linear Programming auf ein Boolean-Satisfiability-Problem reduziert werden. Genauso wie das SAT-Problem ist dieses Problem also NP-Komplex. Um zu beweisen, dass es für dieses Problem keine Lösung gibt, die die Bedingungen erfüllt, müssen alle Kombinationen ausprobiert werden. Somit befindet sich die Umkehrung dieses Problems in der Klasse co-NP.

Einzigartigkeit

Wurde eine Lösung gefunden ist diese auch die einzige mögliche Lösung. Dies ist natürlich nur der Fall, wenn Nachbarn mit der gleichen Größe dedupliziert werden. Die Lösung ist einzigartig, weil das Hinzufügen einer Scheibe an falscher Stelle dazu führt, dass die fälschlicherweise nicht hinzugefügte Scheibe nicht mehr auf den Quader passt, da dieser zu groß geworden ist um sie später noch hinzuzufügen. Welche Scheibe im Fall von mehreren Nachbarn nun die Richtige ist, kann aber nur durch Probieren ermittelt werden, da eine falsche Wahl erst später auffällt, wenn keine Scheiben mehr hinzugefügt werden können und übrig bleiben würden.

Beispiele

Hier werden die Beispiele von der BWINF-Webseite und drei weitere Beispiele bearbeitet.

kaese1.txt

Konsole

Bitte Zahl des Beispiels eingeben: 1

Zeit: 0.00s

Reihenfolge: 2x4 -> 2x4 -> 2x4 -> 3x4 -> 3x3 -> 3x3 -> 3x6 -> 4x6 -> 4x6 ->
4x6 -> 6x6 -> 6x6

Größe: 6x6x6

kaese2.txt

Konsole

Bitte Zahl des Beispiels eingeben: 2

Zeit: 0.00s

Reihenfolge: 998x999 -> 998x999 -> 2x998 -> 2x1000 -> 2x1000

Größe: 2x1000x1000

kaese3.txt

Konsole

Bitte Zahl des Beispiels eingeben: 3

Zeit: 0.00s

Reihenfolge: Zu groß für die Konsole (23 Stück). Siehe output/kaese3.txt.

Größe: 10x1000x1000

output/kaese3.txt

992 995

992 995

2 995

2 993

2 996

:

999 1000

7 1000

1000 1000

1000 1000

1000 1000

kaese4.txt

Konsole

Bitte Zahl des Beispiels eingeben: 4

Zeit: 0.01s

Reihenfolge: Zu groß für die Konsole (550 Stück). Siehe output/kaese4.txt.

Größe: 210x210x210

output/kaese4.txt

```
29 51
29 51
29 51
3 29
3 52
:
208 209
209 209
209 209
209 210
210 210
```

kaese5.txt

Konsole

Bitte Zahl des Beispiels eingeben: 5

Zeit: 0.06s

Reihenfolge: Zu groß für die Konsole (6848 Stück). Siehe output/kaese5.txt.

Größe: 2310x2730x3570

output/kaese5.txt

```
4 1326
4 1326
440 1326
5 1326
5 1326
:
2308 3569
2308 3569
2730 3569
```

```
2309 2730
2730 3570
```

kaese6.txt

Konsole

```
Bitte Zahl des Beispiels eingeben: 6
```

```
Zeit: 1.23s
```

```
Reihenfolge: Zu groß für die Konsole (90300 Stück). Siehe output/kaese6.txt.
```

```
Größe: 30030x39270x510510
```

output/kaese6.txt

```
9255 480255
9255 480255
2 480255
9256 480255
9256 480255
:
30028 39269
39269 510509
30029 510509
30029 39270
39270 510510
```

kaese7.txt

Einen langen Anruf später...

Konsole

```
Bitte Zahl des Beispiels eingeben: 7
```

```
Zeit: 19.01s
```

```
Reihenfolge: Zu groß für die Konsole (1529903 Stück). Siehe
output/kaese7.txt.
```

```
Größe: 510510x510510x510510
```

output/kaese7.txt

```
665 962
665 962
2 962
2 962
667 962
:
510509 510510
510509 510510
510508 510510
510510 510510
510510 510510
```

kaese8.txt

Hier ein eigenes Beispiel zum Testen von aufgegessenen Scheiben.

Konsole

```
Bitte Zahl des Beispiels eingeben: 8

Zeit: 0.04s
Käseblöcke: 1
-----
Käseblock 1:
Reihenfolge: 2x3 -> 1x2 -> 1x2 (wurde aufgegessen) -> 2x5
Größe: 2x2x5
Anzahl aufgegessener Scheiben: 1
```

kaese9.txt

Ein Beispiel zum Testen von mehreren Käseblöcken.

Konsole

```
Bitte Zahl des Beispiels eingeben: 9

Zeit: 0.00s
Käseblöcke: 5
-----
Käseblock 1:
Reihenfolge: 1x1
Größe: 1x1x1
Anzahl aufgegessener Scheiben: 0
-----
Käseblock 2:
```



```
Reihenfolge: 3x3
Größe: 3x3x1
Anzahl aufgegessener Scheiben: 0
-----
Käseblock 3:
Reihenfolge: 5x5
Größe: 5x5x1
Anzahl aufgegessener Scheiben: 0
-----
Käseblock 4:
Reihenfolge: 7x7
Größe: 7x7x1
Anzahl aufgegessener Scheiben: 0
-----
Käseblock 5:
Reihenfolge: 9x9
Größe: 9x9x1
Anzahl aufgegessener Scheiben: 0
```

kaese10.txt

Ein Beispiel zur Kombination von beiden Algorithmen.

Konsole

```
Bitte Zahl des Beispiels eingeben: 10

Zeit: 0.00s
Käseblöcke: 2
-----
Käseblock 1:
Reihenfolge: 4x3 -> 1x3 (wurde aufgegessen) -> 5x3 -> 2x3
Größe: 2x3x6
Anzahl aufgegessener Scheiben: 1
-----
Käseblock 2:
Reihenfolge: 7x7 -> 1x7
Größe: 1x7x8
Anzahl aufgegessener Scheiben: 0
```

Quellcode

program.py

```
import os
import pathlib
import time
```

```

from solve import vanilla, make_stacks

class ExitException(BaseException):
    pass

# pylama:ignore=C901
# Konsolen-Loop
if __name__ == "__main__":
    try:
        while True:
            try:
                # Nutzer nach der Nummer des Beispiels fragen
                num = int(input("Bitte Zahl des Beispiels eingeben: "))
                fname = f"kaese{num}.txt"
                start_time = time.time()
                print()
                # Käsestack Laden
                print(f"Lade {fname}...")
                with open(
                    os.path.join(os.path.dirname(__file__),
f"beispieldaten/{fname}")
                ) as f:
                    stack = [
                        tuple(map(int, f.readline().split()))
                        for _ in range(int(f.readline()))
                    ]
                print("\033[1A\033[2K", end="")
                # Berechnung
                res = None
                try:
                    # Berechnung mit ursprünglicher Methode
                    print("Löse Problem...")
                    res = vanilla(stack)
                finally:
                    print("\033[1A\033[2K", end="")

                # Wenn es kein Ergebnis gibt, versuche es mit der erweiterten
Methode

                if res is None:
                    try:
                        print(
                            "Keine Lösung für einfaches Problem gefunden. "
                            "Verwende erweitertes Problem..."
                        )
                    # Immer mehr Käseblöcke erlauben, bis maximal zur
Länge des Stacks

                    for i in range(1, len(stack) + 1):
                        results = list(make_stacks(stack, i))
                        if len(results) == 0:
                            continue

```

```

        results.sort(key=lambda x: sum(y[2] for y in x))
        res = results[0]
        break
    finally:
        print("\033[1A\033[2K", end="")

# Wenn es immer noch kein Ergebnis gibt, ist das Problem
nicht lösbar
# Das würde heißen es gibt einen Fehler im Programm, da
schlimmstenfalls
# für alle Scheiben ein Käseblock existiert.
if res is None or len(res) == 0:
    raise ExitException("Keine Lösung gefunden.")

# Für einfache Probleme wird nur ein Block ausgegeben
if num in tuple(range(1, 8)):
    path, size, n_virtual = res[0]

    # Speichern der Lösung als Datei
    pathlib.Path(
        os.path.join(os.path.dirname(__file__), "output")
    ).mkdir(parents=True, exist_ok=True)
    with open(
        os.path.join(os.path.dirname(__file__),
f"output/{fname}"), "w"
    ) as f:
        for slice in path:
            f.write(f"{stack[slice][0]} {stack[slice][1]}\n")

    # Ausgabe von Lösungswerten
    print(f"Zeit: {time.time()-start_time:.2f}s")
    print("Reihenfolge: ", end="")
    if len(path) < 10:
        print(
            " -> ".join(
                map(lambda x: f"{stack[x][0]}x{stack[x][1]}",
path)
            )
        )
    else:
        print(
            f"Zu groß für die Konsole ({len(path)} Stück).
Siehe output/{fname}."
        )
    print(f"Größe: {size[0]}x{size[1]}x{size[2]}")
else:
    # Für komplizierte Probleme werden mehrere Blöcke
ausgegeben

    print(f"Zeit: {time.time()-start_time:.2f}s")
    print(f"Käseblöcke: {len(res)}")
    for i, (path, size, n_virtual) in enumerate(res):
        print("-----")
        print(f"Käseblock {i+1}:")

```

```

        print("Reihenfolge: ", end="")
        started = False
        for slice in path:
            if started:
                print(" -> ", end="")
            else:
                started = True
            if isinstance(slice, tuple):
                print(
                    f"{slice[0]}x{slice[1]} (wurde
aufgegessen)", end=""
                )
            else:
                print(f"{stack[slice][0]}x{stack[slice][1]}",
end="")

        print()
        print(f"Größe: {size[0]}x{size[1]}x{size[2]}")
        print(f"Anzahl aufgegessener Scheiben: {n_virtual}")
    # Fehlerbehandlung
    except Exception as e:
        print(f"Fehler: {e}")
    finally:
        print()
except ExitException as e:
    print(e)
    exit()
except KeyboardInterrupt:
    print()
    print("Abbruch durch Benutzer.")
    exit()

```

solve.py

```

import collections
import itertools
from typing import List, Set, Tuple

# pylama:ignore=C901
def vanilla(stack: List[Tuple[int, int]]) -> Tuple:
    """Lösen des Problems mit vollständigem Käsestack und einem Startkäse."""

    # Hashmap für schnellen Zugriff auf potentielle Nachbarn
    lookup = collections.defaultdict(set)
    try:
        for i, (a, b) in enumerate(stack):
            lookup[a].add(i)
            lookup[b].add(i)
    except ValueError:
        raise Exception(
            "ValueError: Die Eingabe-Datei ist wahrscheinlich nicht korrekt

```

```

formatiert."
    )

# Funktion zum Zugriff auf potentielle Nachbarn
def get_neighbors(i: int) -> Set[int]:
    a, b = stack[i]
    ret = (lookup[a] | lookup[b]) - {i}
    return ret

# Liste der Start-Scheiben (sortiert nach Größe, dedupliziert nach Größe)
start_nodes_i = []
seen_start_sizes = set()
for node in list(
    sorted(range(len(stack)), key=lambda i: stack[i][0] * stack[i][1])
):
    size = tuple(sorted(stack[node]))
    if size not in seen_start_sizes:
        start_nodes_i.append(node)
        seen_start_sizes.add(size)

# Besuchte Knoten
seen = set()
# Besucher Pfad mit Größe und zu prüfenden Nachbarn (für backtracking)
path = []
avg_n_neigh = 0
avg_n_neigh_n = 0
while True:
    # Wenn kein Pfad existiert, wähle einen Startknoten
    if not path:
        if start_nodes_i:
            start = start_nodes_i.pop(0)
            path = [[start, tuple(stack[start] + (1,)), None]]
            continue # Diese Iteration überspringen
        else:
            # Es kann keine Lösung gefunden werden
            return

    # Aktueller Knoten, Größe und zu prüfende Nachbarn
    current, size, to_check = path[-1]
    # Füge den aktuellen Knoten zu den besuchten Knoten hinzu
    seen.add(current)
    # Wenn der Pfad vollständig ist, wurde eine Lösung gefunden
    if len(seen) == len(stack):
        return ((list(map(lambda x: x[0], path)), size, 0),)
    # Wenn noch keine Nachbarn geprüft wurden, generiere sie
    if to_check is None:
        to_check = set() # Set für mögliche Nachbarn
        seen_sizes = set() # Set für bereits besuchte Größen
        for i in get_neighbors(current):
            if i in seen: # Nachbarn, die bereits besucht wurden,
                # überspringen
                continue
            ab = set(stack[i])
            new_size = None

```

```

        # Prüfe, ob die Nachbarn die gleiche Größe (in 2 Dimensionen)
haben
        if ab == set(size[:2]):
            new_size = tuple(sorted((size[0], size[1], size[2] + 1)))
        elif ab == set(size[1:]):
            new_size = tuple(sorted((size[1], size[2], size[0] + 1)))
        elif ab == set(size[:,2]):
            new_size = tuple(sorted((size[2], size[0], size[1] + 1)))
        # Wenn zwei mögliche Nachbarn zur gleichen Größe führen,
überspringe den Nachbarn
        if new_size is not None and new_size not in seen_sizes:
            seen_sizes.add(new_size)
            # Füge den Nachbarn zu den zu prüfenden Nachbarn hinzu
            to_check.add((i, new_size))

        # Wenn es keine Nachbarn gibt, entferne den aktuellen Knoten aus dem
Pfad (backtracking)
        if not to_check:
            path.pop()
            seen.remove(current)
            continue
        else:
            avg_n_neigh += len(to_check)
            avg_n_neigh_n += 1
        # Aktualisiere die zu prüfenden Nachbarn im Pfad
        # (nötig wenn die Nachbarn gerade generiert wurden)
        path[-1][2] = to_check
        # Der nächste Knoten ist der erste Nachbar, der noch nicht besucht
wurde
        next_ = to_check.pop()
        path.append([next_[0], next_[1], None])

def covers(it1, it2) -> Tuple | None:
    """Prüft, ob eine Scheibe auf eine andere Scheibe passt.
    Auch Unterschiede von 1 in einer Dimension sind erlaubt.
    Gibt eine Maske von hinzugefügten Scheiben zurück."""
    it1_ = tuple(sorted(it1))
    masks = ((0, 0), (0, 1), (1, 0))
    for mask in masks:
        if tuple(sorted(map(lambda x: x[0] + x[1], zip(it2, mask)))) == it1_:
            return mask

def create_virtual(ab: List, mask: Tuple, ignoredsize: int) -> Tuple or None:
    """Erstellt eine aufgegessene (virtuelle) Scheibe aus einer Scheibe und
    einer Maske."""
    if not any(mask): # Nicht nötig eine virtuelle Scheibe zu erstellen
        return

    # Länge in die Dimension, die nicht passend sein muss x Länge der
    passenden Dimension
    return (ignoredsize, ab[mask.index(0)])

```

```

def remove(slice: Tuple[int, int], size: List[int]):
    """Verändere `size` indem eine Scheibe `slice` entfernt wird."""
    ab = set(slice)
    if ab == set(size[:2]):
        size[2] -= 1
    elif ab == set(size[1:]):
        size[0] -= 1
    elif ab == set(size[::2]):
        size[1] -= 1
    else:
        raise Exception("what")

# pylama:ignore=C901
def fuzzy(stack: List[Tuple[int, int]]):
    """Lösen des Problems mit fehlerhaftem Käsestack und mehreren
    Käseblöcken.
    Gibt eine Liste von möglichen Lösungen zurück."""

    # Dictionary der Lösungen, ihrer Größe und der Anzahl der
    # hinzugefügten/aufgegessenen Scheiben
    solutions = {}

    # Hashmap für schnellen Zugriff auf potentielle Nachbarn
    lookup = collections.defaultdict(set)
    for i, (a, b) in enumerate(stack):
        lookup[a].add(i)
        lookup[b].add(i)

    # Funktion zum Zugriff auf potentielle Nachbarn
    def get_neighbors(i: int) -> Set[int]:
        a, b = stack[i]
        ret = (lookup[a] | lookup[b] | lookup[a + 1] | lookup[b + 1]) - {i}
        return ret

    # Liste der Start-Scheiben (sortiert nach Größe, dedupliziert nach Größe)
    start_nodes_i = []
    seen_start_sizes = set()
    for node in list(
        sorted(range(len(stack)), key=lambda i: stack[i][0] * stack[i][1])
    ):
        size = tuple(sorted(stack[node]))
        if size not in seen_start_sizes:
            start_nodes_i.append(node)
            seen_start_sizes.add(size)

    # Besuchte Knoten
    seen = set()
    # Besucher Pfad mit Größe und zu prüfenden Nachbarn (für backtracking)
    path = []
    # Anzahl der hinzugefügten/'aufgegessenen' Scheiben
    n_virtual = 0

```

```

while True:
    # Wenn kein Pfad existiert, wähle einen Startknoten
    if not path:
        if start_nodes_i:
            start = start_nodes_i.pop(0)
            path = [[0, start, tuple(stack[start] + (1,)), None]]
            continue # Nächste Iteration
        # Wenn keine Startknoten mehr existieren, sind alle Lösungen
        # gefunden
        # und können zurückgegeben werden
        for path, size, n_virtual, n_nodes in solutions.values():
            yield (path, size, n_virtual, n_nodes)
        return # Ende der Funktion
    # Aktueller Knoten, Größe und zu prüfende Nachbarn, und eingefügte
    # 'aufgegessene' Scheiben
    _, current, size, to_check = path[-1]
    # Füge den aktuellen Knoten zu den besuchten Knoten hinzu
    seen.add(current)
    # Wenn noch keine Nachbarn geprüft wurden, generiere sie
    if to_check is None:
        to_check = set() # Set für mögliche Nachbarn
        seen_sizes = set() # Set für bereits besuchte Größen
        for i in get_neighbors(current):
            if i in seen: # Nachbarn, die bereits besucht wurden,
            # überspringen
                continue
            new_sizes = set() # new_size, virtual
            # Prüfe, ob die Nachbarn die gleiche Größe (in 2 Dimensionen)
            # haben
            ab = list(stack[i])
            if s := covers(ab, size[:2]):
                new_sizes.add(
                    (
                        tuple(sorted(ab + [size[2] + 1])),
                        create_virtual(ab, s, size[2]),
                    )
                )
            if s := covers(ab, size[1:]):
                new_sizes.add(
                    (
                        tuple(sorted(ab + [size[0] + 1])),
                        create_virtual(ab, s, size[0]),
                    )
                )
            if s := covers(ab, size[::2]):
                new_sizes.add(
                    (
                        tuple(sorted(ab + [size[1] + 1])),
                        create_virtual(ab, s, size[1]),
                    )
                )
            # Dedupliziere Nachbarn mit gleicher Größe

```



```

        for new_size, virtual in new_sizes:
            if new_size not in seen_sizes:
                seen_sizes.add(new_size)
                # Füge den Nachbarn zu den zu prüfenden Nachbarn
                if virtual and sorted(virtual) == sorted(stack[i]):
                    virtual = None
                to_check.add((i, new_size, None, virtual))
            # Aktualisiere die zu prüfenden Nachbarn im Pfad
            path[-1][3] = to_check
            # Wenn es keine Nachbarn gibt -> backtracking
            if not to_check:
                # Lösung speichern
                filteredpath_ = tuple(map(lambda x: x[1], filter(lambda x: not
x[0], path)))
                path_ = tuple(map(lambda x: x[1], path))
                solutions[filteredpath_] = min(
                    (
                        (path_, size, n_virtual, len(seen)),
                        solutions.get(path_, (path_, size, n_virtual,
len(seen))),
                    ),
                    key=lambda x: sum(x[1]) * x[2],
                )
            # Alle besuchten Knoten ohne alternativen Nachbarn entfernen
            while path and not path[-1][3]:
                i = path.pop()[1]
                seen.remove(i)
                # remove virtual
                if path and path[-1][0] == 1:
                    path.pop()
                    n_virtual -= 1
            continue # Nächste Iteration

        # Der nächste Knoten ist der erste Nachbar, der noch nicht besucht
        wurde
        next_ = to_check.pop()
        next_node, next_size, next_to_check, virtual = next_
        if virtual: # Wenn eine virtuelle Scheibe hinzugefügt werden muss
            path.append([1, virtual]) # 1, virtual size
            n_virtual += 1
        path.append([0, next_node, next_size, next_to_check])

def make_stacks(stack: List[Tuple[int, int]], n: int):
    """Erstelle `n` Blöcke aus `stack`"""
    # Für jede Kombination aus `n` Pfaden
    for c in itertools.combinations(fuzzy(stack), n):
        # Wird die Anzahl der Überflüssigen Scheiben berechnet
        overflow = sum(x[3] for x in c) - len(stack)
        # Wenn die Pfade insgesamt zu wenige Schieben haben, überspringen
        if overflow < 0:
            continue

```

```

# Für jede Kombination die Pfade zu kürzen
for x in itertools.product(range(overflow + 1), repeat=n):
    if (
        sum(x) != overflow
    ): # sodass die richtige Anzahl an Scheiben entfernt wird
        continue
    try:
        # werden alle Pfade gekürzt
        paths = []
        for i, p in enumerate(c):
            path, size, n_virtual = list(p[0]), list(p[1]), p[2]
            for _ in range(x[i]):
                i = path.pop()
                remove(stack[i], size)
                if path and isinstance(path[-1], tuple):
                    s = path.pop()
                    remove(s, size)
                    n_virtual -= 1

            paths.append((tuple(path), tuple(size), n_virtual))
        # Wenn ein Pfad zu kurz ist, um gekürzt zu werden,
        # wird die nächste Kombination ausprobiert
    except IndexError:
        continue
    # Jetzt wird das Ergebnis überprüft,
    if any(len(p) == 0 for p, _, _ in paths):
        continue
    seen = set()
    for p, _, _ in paths:
        for i in p:
            if isinstance(i, tuple):
                continue
            seen.add(i)
    # und wenn vollständig, zurückgegeben
    if len(seen) == len(stack):
        yield tuple(paths)

```