# Aufgabe 1: Flohmarkt in Langdorf

Teilnahme-ID: 58240

Bearbeiter dieser Aufgabe:
Leandro Conte

10. Februar 2021

## Inhaltsverzeichnis



# Lösungsidee

## Das Problem

Das in der Aufgabenstellung gegebene Problem kann folgendermaßen vereinfacht werden:

Man kann sich die N Voranmeldungen, die eine zeitliche und eine räumliche Dimension haben (Standlänge und Standdauer), auch als N Rechtecke vorstellen. Im Rest der Dokumentation ist mit Rechteck also eine Voranmeldung gemeint. In diesem System soll ein Teil der Rechtecke dann auf einer rechteckigen Fläche angeordnet werden, so dass die Gesamtfläche der angeordneten Rechtecke maximal ist. Die Verfügbare Fläche ist $L_x$ = 1000m breit und $L_y$ =18h-8h = 10h lang. Die Gesamtfläche der angeordneten Rechtecke, die Summe der

Produkte aus Strecke s und Zeit t jedes einzelnen Rechtecks, entspricht den Gesamteinnahmen des Marktes in Euro.

Die gesetzten Rechtecke dürfen sich nicht überlappen oder über die Grenzen der Fläche auf der sie angeordnet werden hinausgehen. Außerdem ist für jedes Rechteck schon die zeitliche Position bestimmt.

## Naive Ansätze

Die Menge A der Anmeldungen besitzt $2^N$ Teilmengen. Eine dieser Teilmengen mit k Elementen besitzt maximal $(L_x)^k$ Anordnungsmöglichkeiten.

Jedes Rechteck mit Breite b kann sich in $1+L_x-b$ Zuständen befinden. Ein Zustand für wenn es nicht angeordnet ist und $L_x-b$ verschiedene Anordnungszustände. Da die Zustände der Rechtecke von einander abhängig sind, müsste eine naive Lösung maximal $(1+L_x-1)^N$ Zustandskombinationen durchsuchen. Eine naive Suche ist also zu langsam.

## Darstellung als Graph

Man kann sich einen Graphen vorstellen, mit einem Knoten für jede Anordnungsmöglichkeit jedes Rechtecks. Für jedes Rechteck gibt es $L_x-b$ Knoten. Zwei Knoten sind genau dann durch eine Kante verbunden, wenn die repräsentierten angeordneten Rechtecke nicht miteinander kollidieren. Das Problem ist nun den kompletten Teilgraphen, Clique, zu finden, bei der die Summe aller enthaltenen Rechtecksflächen maximal ist. Dieses Problem ist np-vollständig.

## Ähnlichkeit zum 0-1 Knapsack Problem

Ähnlich wie beim 0-1 Knapsack Problem, muss auch bei diesem Problem aus einer gewichteten Menge an Elementen eine Teilmenge ausgewählt werden, deren gesammeltes Gewicht eine bestimmte Größe nicht überschreitet und deren Wert maximal ist. Der hauptsächliche Unterschied zwischen den zwei Problemen besteht darin, dass die Gewichte, bzw. das gesamte System, 2-dimensional ist, statt 1-dimensional. Eine Lösung wie für das 0-1 Knapsack Problem kann nicht angewandt werde, da es mindestens $L_x^{L_y}$ Gewichtszustände geben kann, statt nur $W$ .

## Eine heuristische Lösung

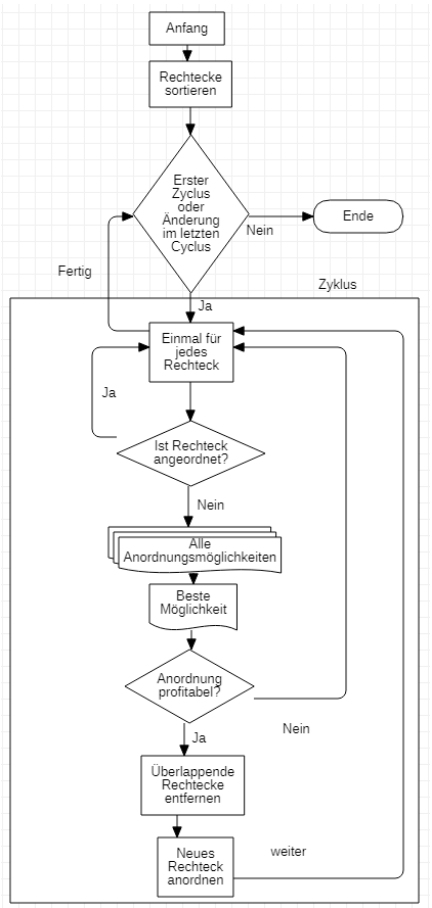
Da alle Ansätze nach einer optimalen Lösung zu np-vollständigen Problemen führen nehme ich an, dass auch das gestellte Problem np-vollständig ist.

Mit einer heuristischen Lösung soll in effizienter Zeit eine möglichst gute Anordnung einer Teilmenge an Rechtecken gefunden werden.

Die Idee ist in mehreren sogenannten „Zyklen“ die Anordnung aufzubauen und zu verbessern. In einem Zyklus wird die gesamte Menge an verfügbaren Rechtecken einmal durchgegangen. A ist

hier die Menge der angeordneten, platzierten Voranmeldungen/Rechtecke und $c(a_1, a_2)$ ist wahr falls die Rechtecke $a_1$ und $a_2$ kollidieren (überlappen).

Das jeweils aktive Rechteck wird dann eingesetzt, wenn es eine Anordnungsmöglichkeit a für das Rechteck gibt, bei der die Fläche der bereits gesetzten Rechtecke, die entfernt werden müssen, um Platz für die neue Anordnung zu machen, für die also gilt $r \in A \wedge c(r, a)$ , kleiner ist als die Fläche des neu eingesetzten Rechtecks.



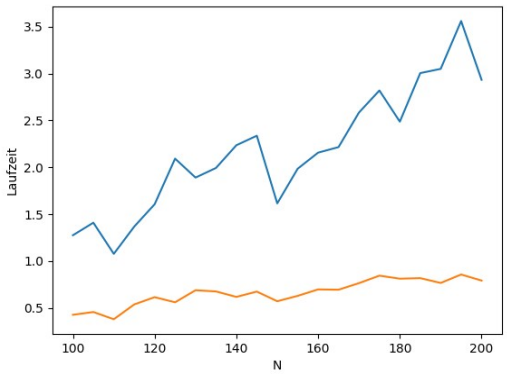

Es soll die Anordnungsmöglichkeit ausgewählt werden, bei der die zu entfernende Fläche minimal ist. Dieser Vorgang neue Rechtecke einzusetzen falls dadurch die Gesamtfläche vergrößert wird, findet so lange statt, bis eine maximale Anzahl an Zyklen erreicht wurde oder die Anordnung im letzten Zyklus nicht verändert wurde, was heißt, dass die Anordnung durch Einsetzen nicht mehr verbessert werden kann.

Durch Modifizieren der Reihenfolge, in der die Rechtecke durchlaufen werden, kann die Qualität des Endergebnisses und die Laufzeit erheblich beeinflusst werden. Wenn die Rechtecke jedes mal von größter Fläche zu kleinster Fläche durchlaufen werden, ist es nach einem Zyklus nicht mehr möglich die Anordnung durch Einsetzen zu verbessern, da ja alle bereits gesetzten Rechtecke nur größer als das neue Rechteck sein können.

Das heißt, das nach einem Durchlauf von groß nach klein, keine neuen Rechtecke mehr eingesetzt werden können.

Alternativ kann die Menge der Rechtecke auch von klein nach groß durchgegangen werden. Diese Methode hat den Vorteil, dass die Rechtecke in bestimmten Fällen besser angeordnet werden und die resultierende Gesamtfläche dann größer ist.

Ein Nachteil ist jedoch, dass die Methode mehrere Zyklen benötigt, was zu einer schlechteren und schwerer zu bestimmenden Laufzeit führt. In allgemeinen Fällen empfiehlt sich deswegen die Sortierung von groß nach klein wegen ihrer besseren Laufzeit. Mehr zur Laufzeit befindet sich im Umsetzungsteil.

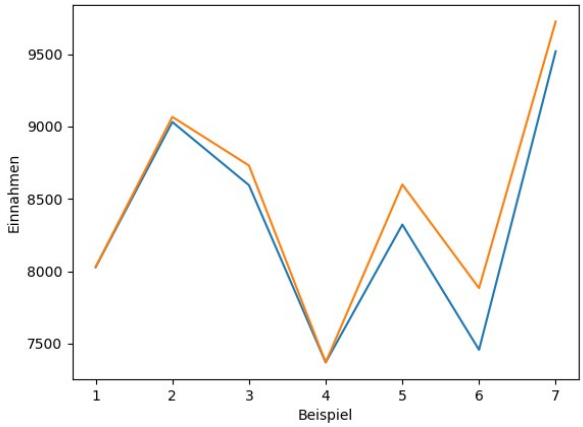



*Wenn von klein nach groß sortiert wird ist die Laufzeit größer: Mehr Zyklen werden benötigt*

### Optimierung der Heuristik

Die bisherige deterministische heuristische Lösung besitzt nur wenige modifizierbare Parameter, weswegen sich das Ergebnis einmal berechnet nur begrenzt optimieren kann, es gibt nur die Wahl der Sortiermethode.

Durch das Einführen eines Zufallsfaktors in die Reihenfolge in der die Rechtecke versucht werden eingesetzt zu werden, kann das Endergebnis optimiert werden. Ein Rechteck wird in jedem Zyklus, außer dem letzten, mit einer Wahrscheinlichkeit p versucht eingesetzt zu werden. Dadurch kann durch wiederholtes Anwenden der modifizierten Verarbeitungsmethode mit unterschiedlichen seeds, eine Menge an Ergebnissen generiert werden, aus der dann das beste Ergebnis ausgesucht wird. Für mehr Laufzeit kann so ein in den meisten Fällen besseres Ergebnis erhalten werden.




*Die "randomisierte" Lösung liefert bei mehreren Wiederholungen bessere Resultate als die einfache Lösung*

# Umsetzung

### Struktur

Ich habe meine Hauptimplementation in zwei Teile geteilt. Dadurch werden die einzelnen Teile übersichtlicher und die Logik einfacher zu erweitern. Zusätzlich befinden sich im Ordner Aufgabe1-Implementierung noch mehrere Erweiterungsdateien. Die Python Implementation verwendet das externe numpy Modul für effizientere Arrays und läuft mit Python Version 3.8.2 (Andere Versionen wurden nicht getestet). Manche Erweiterungen benötigen das matplotlib Modul zur Visualisierung.

### aufgabe1_common.py

Diese Datei enthält allgemeine Strukturen und Funktionen. Darunter ist die *ZoneRequest* Klasse für Voranmeldungen, die *PlacedRequest* Klasse für platzierte Anmeldungen, sowie die *ProposedChange* Klasse für eine Veränderung der Anordnung. Definiert sind auch drei wichtige Konstanten für das Koordinatensystem: *LOWER*, der zeitliche Anfang, *UPPER*, das zeitliche Ende, und *TLEN*, die räumliche Länge. Alle zeitlichen Koordinaten, deren Variablen meist mit t beginnen,

wie *tStart* und *tEnd*, befinden sich im Interval [LOWER; UPPER]. Alle räumlichen Koordinaten befinden sich im Intervall [0; TLEN].
Die Eingabe wird durch die *readInput* Funktion gelesen und in einem *TaskInput* Objekt zurückgegeben. *TaskInput* Objekte enthalten neben der eigentlichen Eingabe, in Form einer Liste an *ZoneRequest* Objekten, auch mehrere Verarbeitungs-Parameter. Diese erkläre ich zusammen mit der Verarbeitungsfunktion für die sie gelten. Alle Verarbeitungs-Parameter, die die Funktion erhalten kann, sind in *TaskInput* enthalten. Die Ausgabe in Form eines *TaskOutput* Objekts kann von der *writeOutput* Funktion geschrieben werden. Die Funktion *doCollide* überprüft, ob zwei *PlacedRequest*s kollidieren und die Funktion *checkNoCollisions* überprüft, ob eine Menge an Platzierungen gültig ist, also frei von Kollisionen. Letztlich stellt *aufgabe1_common* noch einen decorator *randomized* zur Verfügung. Dieser wird auf die Verarbeitungsfunktion angewandt und wiederholt sie mit unterschiedlichen seeds. Die Eingabeparameter *maxRepetitions* und *maxRuntime* (beide in *TaskInput* enthalten) regulieren die Anzahl an Wiederholungen. Falls gegeben versucht *randomized* eine Laufzeit von *maxRuntime* zu erhalten.

## aufgabe1.py

Die Verarbeitungsfunktion *process_standard* befindet sich in dieser Datei. Auf diese wird der *randomized* decorator angewandt, um randomisierte Verarbeitung zu ermöglichen. Der 1. Schritt der Verarbeitung ist das Sortieren der gegebenen Liste an *ZoneRequest* Objekten. Falls der Parameter *reverseRequests* wahr ist wird von größter Fläche zu kleinster sortiert, ansonsten anders herum. In einer while Schleife werden dann die Zyklen ausgeführt, bis eine Grenze erreicht wurde oder nichts mehr verbessert werden kann. Wenn ein Zufallsfaktor benutzt wird, findet der letzte Zyklus immer ohne Zufallsfaktor statt. Pro Zyklus werden alle Anmeldungen durchgegangen. Bei aktiviertem Zufallsfaktor können manche Anmeldungen übersprungen werden, auch wenn sie nicht platziert sind. Für eine Anmeldung werden alle möglichen Platzierungen versucht. Die lokale Funktion *findColliders* gibt für eine Platzierung kollidierende (überlappende) Rechtecke zurück.

Die Implementation benutzt zwei verschiedene Arten platzierte Rechtecke zu speichern. Jedes platzierte Rechteck wird im set *placedRequests* gespeichert und kann auch zusätzlich in einem Array gespeichert werden, mit einer Referenz pro belegtem Raumquadrat. *findColliders* greift hybrid auf eine von zwei Funktionen zurück:

*findInSet* durchsucht alle Elemente in *placedRequests* und testet, ob sie mit der neuen Platzierung kollidieren. Die Laufzeit entspricht ungefähr der Länge von *placedRequests*.

*findInArray* durchsucht alle Einträge des numpy-Arrays *zArray*, welche die neue Platzierung belegen wird. Die Laufzeit entspricht ungefähr der Fläche der neuen Platzierung.

*findColliders* vergleicht die Länge von *placedRequests* mit der Fläche der Platzierung und wählt je nachdem die bessere Suchfunktion aus. Dieser Vergleich könnte noch verfeinert werden, funktioniert aber auch im jetzigen Zustand gut.

Durch diesen hybriden Ansatz ist die Laufzeit praktisch um einiges besser als ohne. Beispiele 4-6 profitieren von *findInSet* und Beispiele 1-3, 7 von *findInArray*.

Nachdem die beste Anordnungsänderung *bChange* gefunden wurde wird sie mit *applyChange* angewandt. Dabei werden *placedRequests* und *zArray* aktualisiert. Optional wird bei jeder Änderung die unter *onChange* erhaltene Callback-Funktion aufgerufen.

Am Ende der Verarbeitung, vor der Ausgabe, wird noch mit *checkNoCollisions* geprüft, dass die Ausgabe gültig ist.

## Laufzeit

Die exakte Laufzeit-Komplexität ist schwer zu schätzen, da so viele Einzelteile involviert sind. Sicher ist, dass, wenn von groß nach klein sortiert wird, die Laufzeit polynomial ist, da die while Schleife maximal 2-mal ausgeführt wird und der Rest des Programms nur einfache for Schleifen und keine Rekursion verwendet. In diesem Fall ist eine obere Grenze für die Laufzeit ungefähr $O\left(N \cdot \left(L_x \cdot R_{findColliders} + R_{applyChange}\right)\right)$ , wo R für die polynomiale Laufzeit der jeweiligen Funktion steht.

Wenn von klein nach groß sortiert wird ist die Laufzeit unvorhersehbarer, da die while Schleife theoretisch sehr lange laufen kann (Jedoch nicht unendlich lange). In beinahe allen Fällen sinkt die Anzahl der möglichen Änderungen mit jedem Zyklus aber schnell, so dass oft schon nach 3 Zyklen nichts mehr verbessert werden kann und die while Schleife beendet wird.

Wenn die randomisierte Version verwendet wird werden auch potentiell mehrere Zyklen benötigt. Die Anzahl an Zyklen kann mit *maxCycles* begrenzt werden.

## aufgabe1_visualizer.py

Die Klasse Visualizer ermöglicht die Visualisierung der Rechtecke. Dazu werden zwei weitere Dateien aufgabe1_visualizer_t.py und aufgabe1_visualizer_m.py benötigt. Die erste Datei verwendet das *turtle* Modul zur Visualisierung und die zweite Datei das *matplotlib* Modul. Falls beide Implementationen verfügbar sind wird die *matplotlib* Implementation verwendet, da sie auch eine interaktive, aktualisierbare, Visualisierung ermöglicht.

## aufgabe1_tester.py

Diese Datei enthält verschiedene Funktionen, die die Laufzeit und Ergebnisqualität der Verarbeitungsfunktion unter Änderung bestimmter Parameter testen und als Graph ausgeben. Hier sind auch die Benchmarks der zwei Erweiterungen gespeichert.

Eine besondere Funktion ist die *showChanges* Funktion, welche für jedes Beispiel die Lösung visuell ausgibt und bei jeder Änderung während der Verarbeitung aktualisiert. So wird die Entwicklung der Anordnung während der Verarbeitung dargestellt. Im Ordner Anhänge befinden sich zwei Videos 0-1.mp4 und 1-0.mp4. Bei 0-1.mp4 wurden die Rechtecke von klein nach groß sortiert, bei 1-0.mp4 anders herum.

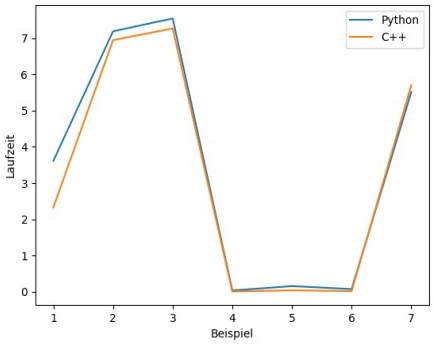
### aufgabe1_generator.py

*generateCase* generiert zufällige Eingaben. Falls dense=True, sind die Rechtecke alle klein, ähnlich wie bei Beispielen 1-3 und 7.

# Erweiterungen

### Implementation in C++11

Obwohl die Laufzeit von 3-7 Sekunden pro Verarbeitung ohne Zufallsfaktor für die größeren Beispiels-Eingaben noch schnell genug für praktische Anwendungen ist, habe ich mich gefragt um wie viel die Laufzeit verbessert werden könnte, wenn die Lösungsidee in einer anderen Sprache implementiert wird. C++ ist im Gegensatz zu Python für seine Geschwindigkeit bekannt. Ich habe die Dateien aufgabe1_common.py und aufgabe1.py mit Ausnahme des randomized Decorators in aufgabe1_common.cpp und aufgabe1_main.cpp übersetzt. Erstaunlicherweise unterscheiden sich die Laufzeiten, wie man rechts sieht kaum. Es ist jedoch wichtig anzumerken, dass die C++ Version nicht optimiert ist, während die Python Version wiederholt überarbeitet wurde.




### Auswahl der Platzierungsreihenfolge durch ein neuronales Netzwerk

Zwei Elementare Entscheidungen, die die bisherige Heuristik „greedy“ löst, sind die Ordnung in der die Anmeldungen durchgegangen und die Auswahl der Platzierung/Änderung, falls mehrere Änderungen lokal gleich gut sind. Meine Erweiterungsidee ist zwei neuronale Netze so zu trainieren, dass sie diese zwei lokale Entscheidungen in Hinsicht auf ihre globale Auswirkung besser treffen können als die bisherige Heuristik. Eine Beispiels-Implementation dieser Idee mit Tensorflow und Keras befindet sich in der Erweiterungsdatei aufgabe1_guided.py. Das Verarbeitungs- und Trainingsverfahren sind in Aufgabe1_Erweiterung.pdf dokumentiert.

# Beispiele

Die Eingaben 1-7 wurden mit einem Zufallsfaktor von 0.6, einer Laufzeitgrenze von 60s und groß nach klein Sortierung verarbeitet. Für alle Ausgaben 1-7 ist verbose=True.

Beispiel 1:

**Eingabe:** flohmarkt1.txt

**Ausgabe:**

Laufzeit: 59.343148500000005s

Einnahmen: 8028€

Standplätze: 490

[490 Zeilen folgen. Siehe ausgabe1.txt]

## Beispiel 2:

**Eingabe:** flohmarkt2.txt

**Ausgabe:**

Laufzeit: 58.2418868s

Einnahmen: 9060€

Standplätze: 498

[498 Zeilen folgen. Siehe ausgabe2.txt]

## Beispiel 3:

**Eingabe:** flohmarkt3.txt

**Ausgabe:**

Laufzeit: 57.09741150000001s

Einnahmen: 8721€

Standplätze: 579

[579 Zeilen folgen. Siehe ausgabe3.txt]

## Beispiel 4:

**Eingabe:** flohmarkt4.txt

**Ausgabe:**

Laufzeit: 59.99179940000002s

Einnahmen: 7370€

Standplätze: 5

8 12 249 420

8 9 420 923

8 15 0 249

15 18 0 526

9 15 420 897

## Beispiel 5:

**Eingabe:** flohmarkt5.txt

**Ausgabe:**

Laufzeit: 59.911585900000034s

Einnahmen: 8599€

Standplätze: 7

14 18 526 927

8 12 0 714

8 12 714 908

9 16 936 967

12 15 0 520

8 10 908 912

15 18 0 526

## Beispiel 6:

**Eingabe:** flohmarkt6.txt

**Ausgabe:**

Laufzeit: 59.9666191s

Einnahmen: 7883€

Standplätze: 6

15 18 0 526

8 12 0 171

9 15 171 648

8 9 171 674

8 15 674 923

12 15 0 171

## Beispiel 7:

**Eingabe:** flohmarkt7.txt

**Ausgabe:**

Laufzeit: 61.2546059s

Einnahmen: 9749€

Standplätze: 531

[531 Zeilen folgen. Siehe ausgabe7.txt]

[Beispiele einfügen]

## Beispiel 8:

**Eingabe:** beispiel8_ein.txt

**Ausgabe:** beispiel8_aus.txt

144.8382473

9934

109

[109 Zeilen folgen. Siehe beispiel8_aus.txt]

## Beispiel 9:

**Eingabe:** beispiel9_ein.txt

**Ausgabe:** beispiel9_aus.txt

24.378010899999992

9954

6

[6 Zeilen folgen. Siehe beispiel9_aus.txt]

Zu Beispiel 8-9:

Die Beispiele zeigen, dass *process_standard* selbst bei klein nach groß Sortierung und größeren Ns (N=1000) in weniger als 3 Minuten läuft. Dies gilt auch bei Eingaben mit vielen kleinen Rechtecken, wie Beispiel 8.

# Quellcode

```
@randomized
def process_standard(tInput: TaskInput, onChange: Callable[[ProposedChange], None] = None) -> TaskOutput:
    """
    Processes input with heuristic method, dynamic structures and optional randomization
    """
    def yesorno(yesProb: float):
        return rnd.random() < yesProb


    def findInSet(nPlaced: PlacedRequest) -> Tuple[List[PlacedRequest], int]: #O(len(placedRequests))
        """Finds collision by iterating through set"""
        tArea = 0
        colliders = []
        for pReq in placedRequests:
            if doCollide(pReq, nPlaced):
                tArea += pReq.area
                colliders.append(pReq)
        return colliders, tArea


    def findInArray(nPlaced: PlacedRequest) -> Tuple[List[PlacedRequest], int]: #O(tLen*zLen)
        """
        Finds collisions by iterating through array
        """
        tArea = 0
        colliders = set()
        for col in range(nPlaced.tStart-LOWER, nPlaced.tEnd-LOWER):
            for row in range(nPlaced.zStart, nPlaced.zEnd):
                pReq = zArray[col, row]
                if pReq is not None and not pReq in colliders:
```

```
            colliders.add(pReq)
            tArea += pReq.area
    return list(colliders), tArea


def findColliders(nPlaced: PlacedRequest) -> Tuple[List[PlacedRequest], int]: #O(min(len(placedRequests), tLen*zLen))
    if tInput.forceSet:
        return findInSet(nPlaced)
    elif tInput.forceArray:
        return findInArray(nPlaced)
    elif len(placedRequests)  < nPlaced.area:
        return findInSet(nPlaced)
    else:
        return findInArray(nPlaced)

def applyChange(change: ProposedChange): #O(len(colliders)*col.tLen*col.zLen)
    nonlocal didChange
    if onChange is not None: #Call hook
        onChange(change)
    #Remove colliding placements
    for collider in change.colliders:
        if not tInput.forceSet:
            zArray[collider.tStart-LOWER:collider.tEnd-LOWER, collider.zStart:collider.zEnd] = None
        placedRequests.remove(collider)
        isPlaced[collider.localID] = False
    #Add new placement
    nPlaced = change.nPlaced
    if not tInput.forceSet:
        zArray[nPlaced.tStart-LOWER:nPlaced.tEnd-LOWER, nPlaced.zStart:nPlaced.zEnd] = nPlaced

    placedRequests.add(nPlaced)
    isPlaced[nPlaced.localID] = True
    didChange = True

sTime = timeit.default_timer()
if tInput.useRandomization:
    rnd = random.Random()
    if tInput.seed == -1:
        rnd.seed()
```

```
    else:
        rnd.seed(tInput.seed)

requests = tInput.requests.copy()
requests.sort(key=areaSort, reverse=tInput.reverseRequests)

placedRequests: Set[PlacedRequest] = set()

if tInput.forceArray or not tInput.forceSet:
    zArray = np.full(shape=(UPPER-LOWER, TLEN), dtype=PlacedRequest, fill_value=None)

isPlaced: List[bool] = [False for _ in requests]

for i in range(len(requests)):
    requests[i].localID = i

didChange = True
finalRound = False
cycleCount = 0
while (didChange or finalRound) and (tInput.maxCycles == -1 or cycleCount < tInput.maxCycles):
    cycleCount += 1
    didChange = False

    for req in requests:
        if isPlaced[req.localID]:
            continue
        if tInput.useRandomization:
            if not finalRound and not yesorno(tInput.considerElementProb):
                continue

        bChange: Union[ProposedChange, None] = None

        for zStart in range(TLEN-req.zLen):
            nPlaced = PlacedRequest(req, zStart)
            colliders, tArea = findColliders(nPlaced)
            if tArea < req.area and (bChange is None or bChange.addedArea < nPlaced.area-tArea):
                bChange = ProposedChange(colliders, nPlaced, tArea)

        if bChange is not None:
```

```
            applyChange(bChange)

    if tInput.useRandomization:
        if not didChange and not finalRound:
            finalRound = True
        else:
            finalRound = False

assert checkNoCollisions(placedRequests)
tOutput = TaskOutput(placedRequests)
tOutput.runtime = timeit.default_timer()-sTime
return tOutput
```