

Fortgeschrittene(re) Algorithmen

Übung 12

Einführung in die Programmierung

Über dieses Übungsblatt

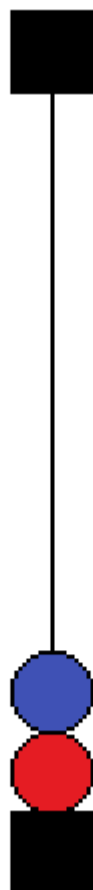
Im diesem Blatt vertiefen wir unser Verständniß von fortgeschrittenen Algorithmen. Sie können Numpy für das ganze Blatt verwenden, wenn Sie das möchten.

Aufgabe Bonus: Newton'sche Physik 3

Letzte Änderung: 06. July 2023, 13:26 Uhr

10 Punkte — [im Detail](#)

Ansicht:  | 



Das zu simulierende Objekt

Wir möchten nun unsere Physiksimation der letzten Woche erweitern. Eine Musterlösung der letzten Woche und animiertes gif der Lösung liegen in LMS bereit.

Wir möchten nun eine zweite Kugel hinzufügen; die erste Kugel startet wie gewohnt ganz oben, die zweite Kugel startet in der Mitte des Drahts. Beide Kugeln besitzen getrennte Geschwindigkeitswerte, fallen aber mit der gleichen Beschleunigung nach unten. Beide Kugeln können die Begrenzungen der "Stopper" oben und unten weiterhin nicht überschreiten. Die obere Kugel kann auf der unteren Kugel liegen wie oben gezeigt, aber nie niedriger fallen (die Kugeln dürfen sich also nie überschneiden).



Man kontrolliert nun die untere Kugel mit der "u" Taste wie in Aufgabenteil 5). Wenn die untere Kugel die obere Kugel berührt bzw. "anstößt", entsteht eine sogenannte [elastische Kollision](#). Wenn beide Objekte die gleiche Masse haben (was wir hier annehmen), **tauschen beide Objekte bei einer Kollision einfach ihre Geschwindigkeit**.

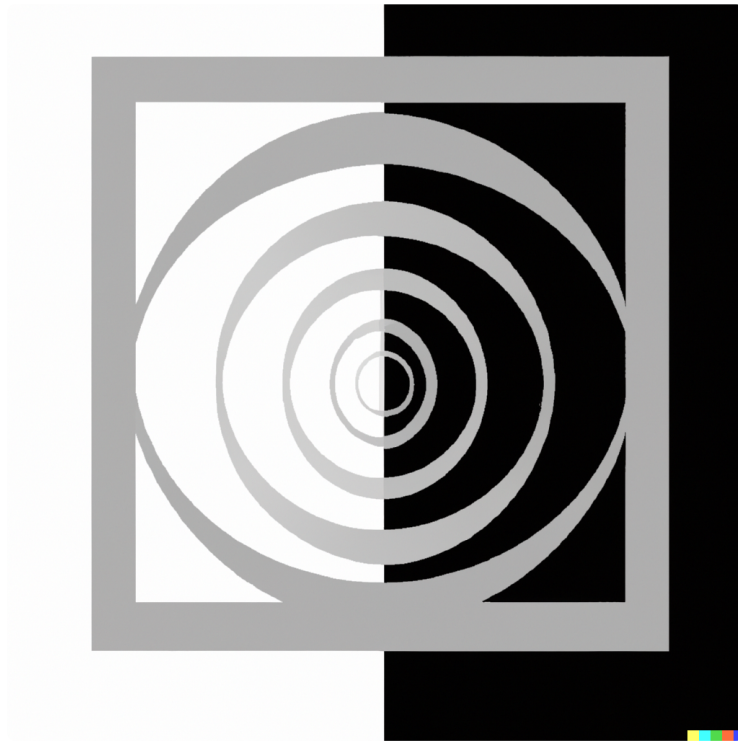
6) Programmieren Sie die oben beschriebene Erweiterung der Physiksimulation.

Aufgabe Verständnisfragen

Letzte Änderung: 06. July 2023, 13:26 Uhr

5 Punkte — [im Detail](#)

Ansicht:  | 



"An artwork that is a visual illustration of recursion", generiert mit [Dalle 2](#)

Zum besseren Verständnis, zunächst eine paar Theoriefragen. Antworten Sie mit ja/nein und begründen Sie ihre Antwort.

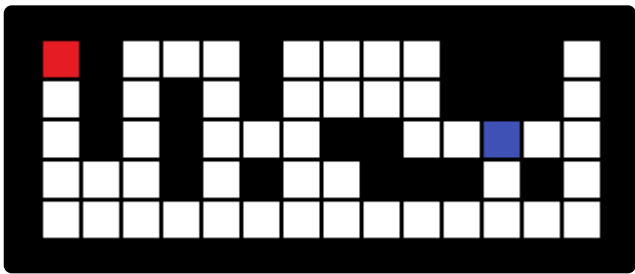
- 1a) Im schlimmsten Fall muss ein backtracking Algorithmus alle gültigen Kombinationen / alle mögliche Lösungen im Problemraum aufzählen und hat somit eine exponentielle (also sehr langsame) Laufzeit.
- 1b) Ein backtracking Algorithmus kann nicht iterativ geschrieben werden.
- 1c) Ein backtracking Algorithmus eignet sich besonders für schwere Probleme, also Probleme bei denen sich die Menge aller möglichen Lösungen als baumartige Struktur darstellen lässt.
- 1d) Backtracking ist ein rekursiver Algorithmus, deswegen ist es unsinnig darin **for**-Schleifen zu verwenden.
- 1e) Backtracking und branch and bound Algorithmen unterscheiden sich fundamental nur durch einer zusätzliche **if**-Abfrage.

Aufgabe Labyrinth: Branch and Bound

Letzte Änderung: 06. July 2023, 13:26 Uhr

10 Punkte — [im Detail](#)

Ansicht:  | 



Ein Labyrinth mit mehreren Wegen zum Ziel. Wie findet man den kürzesten?

In dieser Aufgabe möchten wir unseren backtracking Algorithmus von letzter Woche verbessern. Eine Musterlösung der letzten Woche und ein Labyrinth mit mehreren Pfaden zum Ziel liegt in LMS bereit.

1) Verändern Sie den backtracking Algorithmus so, dass er nicht nur *einen* Pfad durch das Labyrinth findet, sondern alle.

Tipp: Sie können eine globale Variable am Anfang definieren und über alle Funktionsaufrufe hinweg nutzen.

```
global_list = []

def backtrack(...):
    global global_list
    # now we can use global list in this function
```

2a) Fügen Sie eine zusätzliche Bedingung *vor* den rekursiven Aufruf hinzu: dieser soll nur dann erfolgen, wenn die Pfadlänge der aktuellen Lösung kleiner ist als die kleinste Pfadlänge aller bisher gefundenen Lösungen (also Pfade von Eingang bis Ausgang). Das Prinzip von **branch and bound** ist einfach: wenn wir sowieso schon eine bessere Lösung haben, müssen wir den Lösungskandidaten nicht weiterverfolgen.

Tipp: Sie können die *copy*-Funktion nutzen um die beste gefundene Lösung zu aktualisieren:

```
from copy import copy

best_sol = copy(current_sol)
```

2b) Was passiert durch die in 2a) vorgenommene Modifikation mit dem Baum aller möglichen Lösungen, den wir mithilfe unseres Algorithmus durchlaufen?

Aufgabe Subset Sum

Letzte Änderung: 06. July 2023, 13:26 Uhr
20 Punkte — [im Detail](#)

Ansicht: |

j/v		0	1	2	3	4	5	6	7	8	9	10	11	12	13
3		0	0	0	0	0	0	6	6	8	8	8	8	8	12
2		0	0	0	0	0	0	6	6	8	8	8	8	8	8
1		0	0	0	0	0	0	0	0	8	8	8	8	8	8
0		0	0	0	0	0	0	0	0	0	0	0	0	0	0

Das Teilsummenproblem zum Beispiel (siehe Unten) gelöst durch dynamisches Programmieren in Excel

Das [Teilsummenproblem](#) (engl. subset sum) ist eine "Vereinfachung" des bekannten [Rucksackproblems](#) (engl. knapsack), bei dem der Wert der gepackten Objekten deren Gewicht entspricht. Als Motivation könnte man sich folgendes Szenario vorstellen: Sie haben eine CPU mit W freien Zyklen und möchten die Menge der Aufträge (die jeweils w_i Zeit benötigen) auswählen, die die Anzahl von Leerlaufzyklen minimiert.

Gegeben ist eine Liste **weights** von Gewichten (Fließkommazahlen, entsprechen den w_i) von Gegenständen sowie eine Zahl **max_weight** (Fließkommazahl > 0 , entspricht W), die das maximal erlaubte Gesamtgewicht angibt. Ihre Aufgabe ist es, das maximale Gewicht an

Gegenständen zu berechnen, die man einpacken kann, ohne das Gesamtgewicht zu überschreiten, und die maximal viel Gewicht einpackt (den Rucksack also so voll wie möglich macht).

Beispiel: `weights = [8, 6, 6]` sowie `max_weight = 13` führt zu Ergebnis `12` (zweiten und dritten Gegenstand einpacken).

Wir werden das Problem in drei Effizienzstufen lösen und dabei die Laufzeiten von jeder der drei Lösungen messen. Nutzen Sie dafür die `time`-Klasse:

```
import time

start = time.time()
print("hello")
end = time.time()
print(end - start)
```

und verwenden Sie den folgenden Benchmark für die Zeitmessung:

```
weights = [8, 15, 1, 14, 77, 13, 15, 21, 13]
max_weight = 96

# correct solution is 93
```

1a) Level 1: Lösen Sie das Problem durch das Ausprobieren **aller** Kombinationen von Gewichten.

Tipp: Dies können Sie z.B. folgendermaßen realisieren: schreiben Sie einen backtracking Algorithmus, aber anstatt die Validität der Teillösung *vor* dem rekursiven Aufruf zu prüfen, prüfen Sie sie erst ganz *am Ende*, wenn die Lösung vollständig ist (Rekursionsanfang). Dies ist absolut eine blöde Idee, uns geht es aber darum zu verstehen warum.

1b) Messen Sie die Laufzeit ihres Algorithmus.

2a) Level 2: Lösen Sie das Problem mit einem backtracking Algorithmus.

2b) Messen Sie die Laufzeit ihres Algorithmus. Welche Funktionsaufrufe haben Sie eingespart?

3a) Level 3: Durch die zusätzliche Annahme, dass W und w_i ganze Zahlen sind, können wir einen (in manchen Fällen) noch effizienteren Algorithmus durch dynamisches Drogrammieren gestalten. Beim dynamischen Programmieren wenden wir wieder das klassische "divide-and-conquer Prinzip" an. Die Rekurrenzgleichung folgt im Dynamischen programmieren meist aus einer ganz einfachen Aussage (man könnte es ironisch auch "no shit statement" nennen); in unserem Beispiel konkret:

- Entweder die optimale Lösung enthält ein Gewicht w_i , oder eben nicht.

Diese zugegebenermaßen einfache Aussage kann man weiter formalisieren. Sei $OPT(j, V)$ die Lösung unseres Problems für alle Gewichte *bis index* j für ein Maximalgewicht V . Sei n die Gesamtanzahl an Gewichten. Fangen wir also beim letzten Gewicht an: entweder es ist in der optimalen Lösung enthalten und es gilt somit $OPT(n, W) = w_n + OPT(n - 1, W - w_n)$ oder nicht und es gilt $OPT(n, W) = OPT(n - 1, W)$. Man kann also ganz allgemein für jedes $j \leq n$ und $w_j \leq V \leq W$ sagen:

$$OPT(j, V) = \max(OPT(j - 1, V), w_j + OPT(j - 1, V - w_j)).$$

Da V eine Zahl zwischen 0 und W ist und j eine Zahl zwischen 0 und n ist, kann man das Problem einfach in eine $W \times n$ große Tabelle Übertragen, die man von unten bis oben und links nach rechts ausfüllt. Eine Zelle dieser Tabelle wird also immer mithilfe zweier Zellen der Zeile darunter berechnet. Die finale Antwort steht dann oben rechts in der Tabelle. Es gelten trivialerweise die Anfangsbedingungen $OPT(0, V) = OPT(j, 0) = 0$ für alle $V \leq W$ und $j \leq n$. Die Illustration am Anfang der Aufgabe zeigt die Lösung für das Beispiel mit den drei Gewichten am Anfang. Ihre Tabelle sollte folgende Form haben:

$OPT(n,0) = 0$	$OPT(n,1)$	$OPT(n,2)$...	$OPT(n,W)$
...
$OPT(2,0) = 0$	$OPT(2,1)$	$OPT(2,2)$...	$OPT(2,W)$
$OPT(1,0) = 0$	$OPT(1,1)$	$OPT(1,2)$...	$OPT(1,W)$

$\text{OPT}(n,0) = 0$	$\text{OPT}(n,1)$	$\text{OPT}(n,2)$...	$\text{OPT}(n,W)$
$\text{OPT}(0,0) = 0$	$\text{OPT}(1,V) = 0$	$\text{OPT}(2,V) = 0$...	$\text{OPT}(0,W) = 0$

Lösen Sie das Problem durch dynamisches Programmieren.

Hinweis: Es gilt auch $\text{OPT}(j,V) = \text{OPT}(j-1,V)$ wenn $w_j > V$, also wenn das Objekt schwerer ist als das Maximalgewicht, also in keinem Fall reinpasst. Diesen Fall müssen Sie separat programmieren.

3b) Messen Sie die Laufzeit ihres Algorithmus und vergleichen Sie diese mit den vorherigen Laufzeiten. Verändern Sie den Wert `max_weight` auf **1000** und messen Sie erneut alle drei Laufzeiten. Was fällt Ihnen auf? Erklären Sie ihre Beobachtungen.