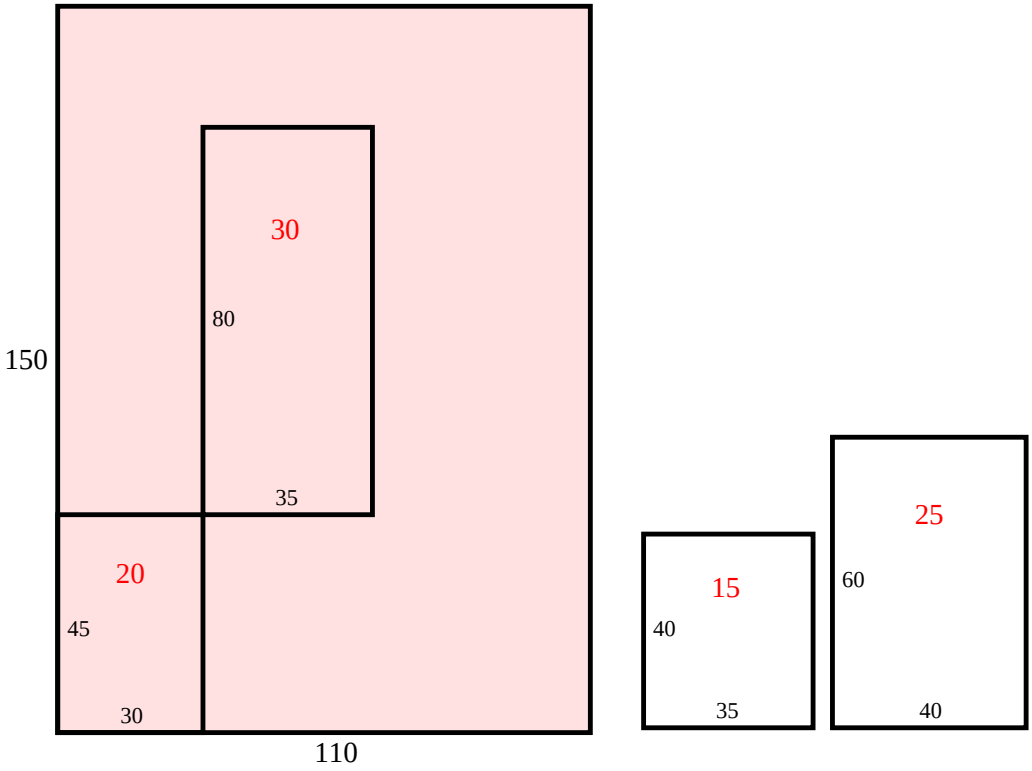
Universiteit van Amsterdam

Programmeertalen

Python

Bas Terwijn, Kyrian Maat

Knapsack



1 Introductie

Bij een knapsack-probleem, zoals weergegeven in figuur 1, krijgen we punten voor elk item dat we inpakken in de knapsack. Elk item mag maar één keer worden ingepakt. De knapsack heeft beperkte resources, bestaande uit 'weight' en 'volume', waardoor niet alle items kunnen worden ingepakt. Het totaal aan resources van de ingepakte items mag de resources van de knapsack niet overschrijden. Welke items in figuur 1 zou jij inpakken om een zo hoog mogelijk puntentotaal in de knapsack te krijgen (zie ook het figuur op het voorblad)?

Figuur 1: visualisatie van het knapsack_small.csv bestand

In deze opdracht zijn er drie verschillende CSV-bestanden aangeleverd die elk een Knapsack-probleem representeren die je gaat oplossen:

- knapsack_small.csv, 4 items
- knapsack_medium.csv, 18 items
- knapsack_large.csv, 1000 items

Deze CSV bestanden zul je tijdens de opdracht inlezen en hebben een duidelijk formaat. Kijk als voorbeeld in het "knapsack_small.csv" bestand welke in Figuur 1 is geïllustreerd.

2 Het opdelen van het probleem: Object-Oriented Design

Maak voordat je gaat programmeren eerst een ontwerp wat een natuurlijke opdeling van het probleem zou moeten geven. Maak een UML class diagram met minimaal de classes:

- Knapsack
- Resources
- Item
- Items
- en later de hieronder genoemde Solvers

Gebruik waar mogelijk dunder methods om het gebruik van operatoren en speciale functies (bijvoorbeeld: +, +=, [ ], len) mogelijk te maken. Pas encapsulatie toe zodat de code buiten een class alleen de methoden (het interface) van de class aanroept en niet direct toegang heeft tot de attributen (de implementatie-details) van de class. Net zoals bij functies is het beter om veel verschillende classes te hebben die ieder hun eigen verantwoordelijkheid hebben dan weinig classes die meerdere verantwoordelijkheden hebben, dus splits een class op wanneer dat redelijkerwijs kan. Vergelijk je ontwerp met anderen.

2.1 UML class diagram

Houd het simpel, we gebruiken hier UML slechts op een informele manier. Een formele UML specificatie met veel details helpt niet voor ons doel hier.

We willen dat je zelf vooraf over de structuur van je code nadenkt en dat je deze structuur gemakkelijk met elkaar kan bespreken. Het diagram is slechts een hulpmiddel om tot een goede code-structuur te komen, als je later besluit om een andere structuur te gebruiken is dat prima, pas dan je diagram aan. Gebruik pen en papier of bijvoorbeeld één van deze tools:

- Umletino
  - instructie: https://youtu.be/3UHZedDtr28
- Dia Diagram Editor (see package manager)
  - instructie: https://youtu.be/f-IeQbc2o5k

3 Het probleem oplossen: Solvers

We maken verschillende solvers om de knapsack-problemen op te lossen. Een solver probeert verschillende manieren van inpakken en onthoudt daarvan de beste. Elk type solver moet aan onderstaande solve() functie mee kunnen worden gegeven, een voorbeeld van polymorfisme. Deze solve() functie is gedefinieerd in het bestand "knapsack.py". Pas de code in dit bestand niet aan, maar voeg alleen je eigen code toe.

```
def solve(solver, knapsack_file, solution_file):
    """ Uses 'solver' to solve the knapsack problem in file
    'knapsack_file' and writes the best solution to 'solution_file'.
    """
    knapsack, items = load_knapsack(knapsack_file)
    solver.solve(knapsack, items)
    knapsack = solver.get_best_knapsack()
    print(f"saving solution with {knapsack.get_points()} points to '{solution_file}'")
    knapsack.save(solution_file)

solve(example_solver,
      "knapsack_small.csv",
      "knapsack_small_solution.csv") # for example
```

De oplossing voor het bestand 'knapsack_small_solution.csv' zou er dan zo uit moeten zien:

```
points:60
item1
item3
item4
```

Houd je aan deze uitvoer! Schrijf dus eerst het puntentotaal op dezelfde manier uit, en schrijf daarna elk item wat in de knapsack is ingepakt op een eigen regel (in een willekeurige volgorde).

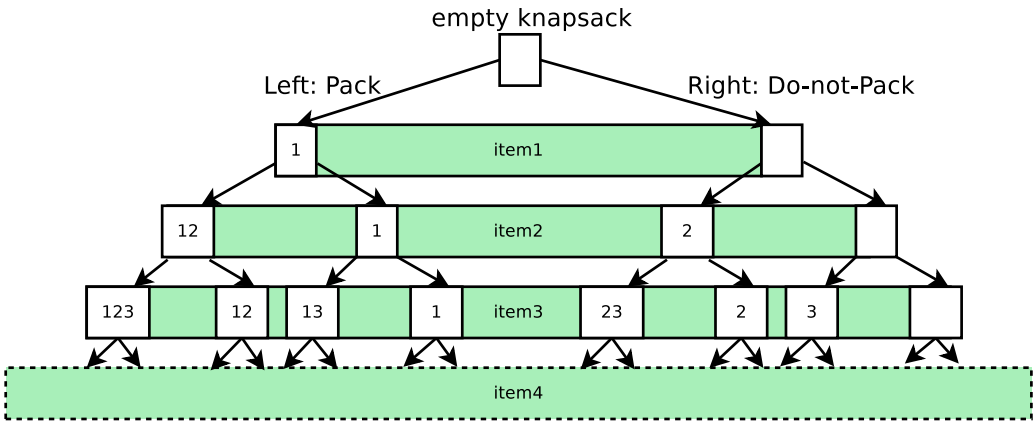
3.1 De eerste stap: Random Solver

We gaan nu alle Solvers in subsecties kort toelichten. Schrijf als eerste een eenvoudige Random_Solver die random items in de Knapsack inpakt totdat er geen item meer bij kan. Deze Solver hoeft nog niet hele goede oplossingen te vinden. De Solver krijgt een argument mee (hier 1000), wat aangeeft hoeveel keer deze solver moet proberen om een lege Knapsack in te pakken. Misschien vindt deze solver met genoeg pogingen wel de best mogelijke oplossing.

```
solver_random = Solver_Random(1000)
```

3.2 Een stap beter: Optimal Solver

Om een optimale oplossing gemakkelijker te vinden kunnen we depth-first alle mogelijke inpakkingen van items af gaan zoals schematisch weergegeven in figuur 2.



Figuur 2: Zoekboom van het Knapsack probleem

3.2.1 Recursieve solver

Schrijf een Solver_Optimal_Recursive solver die dit recursief doet.

```
solver_optimal_recursive = Solver_Optimal_Recursive()
```

3.2.2 Iteratieve solver met deepcopy

Schrijf een Solver_Optimal_Iterative_Deepcopy solver die dit niet recursief maar iteratief doet. Gebruik hiervoor dus een eigen stack en niet de stack die de python runtime environment gebruikt bij het uitvoeren van (recursieve) functies. Het makkelijkste is om hierbij herhaaldelijk kopieën van de knapsack met verschillende inpakkingen op de stack the plaatsen met gebruik van copy.deepcopy()[1]. Dit kost wel veel geheugen en maakt het programma langzamer dan de recursieve implementatie.

```
solver_optimal_iterative_deepcopy = Solver_Optimal_Iterative_Deepcopy()
```

3.2.3 Iteratieve solver zonder deepcopy

De recursieve implementatie is sneller omdat het geen kopieën van de knapsack op de stack plaatst maar in plaats daarvan items die wel of niet ingepakt moeten worden. Schrijf een snelle Solver_Optimal_Iterative solver die iteratief ook, net als de recursieve implementatie, items op de stack plaatst en niet kopieën van de knapsack. Als je dit moeilijk vindt mag je ook een snelle iteratieve Solver_Optimal_Iterative solver schrijven met gebruik van de itertools.combinations()[2] functie.

```
solver_optimal_iterative = Solver_Optimal_Iterative()
```

3.3 Mijn programma is te traag: Random Improved

Het knapsack_large.csv probleem is veel te groot om met een optimale solver, die alle mogelijkheden afgaat, op te lossen. De Random_Solver kan wel snel oplossingen vinden, maar deze hebben meestal geen hoog puntentotaal. Schrijf een eigen Solver_Random_Improved solver die tot betere oplossingen komt dan de Random_Solver.

[1] Klik voor documentatie over copy.deepcopy()
[2] Klik voor documentatie over itertools.combinations()

```
solver_random_improved = Solver_Random_Improved(5000)
```

Denk daarbij bijvoorbeeld aan deze aanpak (web-search: “hill climbing algorithm”):

```
Pak de knapsack random in
Herhaal:
   Maak een random aanpassing: haal 1 item eruit en probeer andere items toe te voegen
   Maak deze aanpassing ongedaan als het puntentotaal hierdoor verslechtert
```

maar een andere aanpak is ook toegestaan zolang dit een significant beter resultaat geeft dan de Solver_Random.

4 Tips & Tricks

- Zorg voor een duidelijk UML class diagram wat een goed overzicht van de structuur van je code geeft.
- Zorg voor goed leesbare code met makkelijk te begrijpen namen. Object-Orientatie zou hierbij moeten helpen.
- Maak gebruik van encapsulation (geen directe toegang tot attributen van een class, roep in plaats daarvan methoden aan).
- Vaak moet het puntentotaal van een knapsack worden berekend. Het is niet efficiënt om elke keer opnieuw de som van punten van alle items te berekenen. Beter is het om de knapsack zelf een eigen “points” attribute te geven. Dit zal leiden tot een invariantie.
- Zorg dat alle invarianties goed worden afgeschermd door gebruik van encapsulation en een “_” prefix bij namen van attributen. Zie de slides van het hoorcollege voor een voorbeeld.
- Voorkom code duplicatie, bijvoorbeeld door gebruik van inheritance bij de verschillende Solvers.
- Schrijf documentatie voor classes en functies in Docstring formaat.
- Volg de Style Guide zoals gedefinieerd door Flake8. Hier wordt bij inleveren op getest.
- Check de rubric op CodeGrade.

5 Inleveren

Lever de volgende bestanden in:

knapsack.pdf	UML class diagram voor het Knapsack probleem
knapsack.py	Gegeven “knapsack.py” bestand aangevuld met classes voor het Knapsack probleem en de solvers