Multiple Knapsack Problem

Beim Multiple Knapsack Problem (MKP) geht es darum, Items (mit einem Profit und einem Gewicht) in verschiedene Knapsacks einzustaplen. Jeder Knapsack hat aber ein Maximalgewicht, welches nicht unterschritten werden darf. Die Einnahmen (später als *Earnings* bezeichnet) sind die Summe der Profits der mitgenommenen Items. Allersdings erzeugt freier Platz in einem Knapsack eine Strafe, welche pro nicht genutztem Gewicht entspricht. Die Summe der Strafen über alle Knapsacks wird als *Penalty* bezeichnet. Ziel ist es, die Differenz $\text{Earnings} - \text{Penalty} = \text{Profit}$ zu maximieren.

Lösungen des MKP werden in einer Liste dargestellt. Jedes Item hat eine ID, die den Index in dieser Liste darstellt. Den Wert an dieser Stelle entspricht der Knapsack-ID, sodass z.B. folgende Lösung [1, 1, -1, 2, 0] bedeutet, dass folgenden Zuordnung getroffen wurde:

- Knapsack 0: Item mit ID 4
- Knapsack 1: Items mit ID 0 und 1
- Knapsack 2: Item mit ID 3 Das Item mit ID 2 wurde nicht zugeordnet und damit auch in keinen Knapsack gepackt.

Diese Repräsentation hat den Vorteil, dass man möglichst viel von der Codebasis aus dem Seminar benutzen kann, man muss nur kleinere Änderungen machen. Insbesondere habe ich überall Type Hints hinzugefügt und eine Klassen zu Dataclasses gemacht, was das Weiterbenutzen des Codes viel leichter macht und den Code auch verständlicher macht. Weiterhin habe ich alle Klassen im Unterverzeichnis `cls` gespeichert.

Ein Lösungsobjekt kann einfach aus der Listendarstellung erzeugt werden und besitzt folgende Methoden und Attribute:

- `allocation` : Die Liste, die die Zuordnungen zwischen Item und Knapsack enthält
- `penalty` : Jedes restliche Volumen im Knapsack wird bestraft. Die Summe der Strafen ist in `penalty` gespeichert.
- `earnings` : Jedes Item ist unterschiedlich viel Wert. Sie Summe der Werte der einzelnen mitgenommenen Items wird in `earnings` gespeichert.
- `profit` : `earnings - penalty`
- `Solution.printSolution()` gibt eine übersichtlichere Darstellung der Lösung aus
- `Solution.to_json(filename: str, includeEarnings = False, includePenalty = False, force = False)` schreibt die Lösung im JSON-Format in die angegebene Datei. Es wird überprüft ob die Datei schon existiert, wenn ja, wird diese nicht überschrieben, es sei denn `force = True` . Die Earnings und Penalties werden normalerweise nicht mit abgespeichert, dies lässt sich mittels `includeEarnings` und `includePenalty` ändern.

In [1]:
```
from cls.Solution import Solution
from cls.InputData import InputData
from cls.EvaluationLogic import EvaluationLogic

import os

data = InputData(os.path.join(os.getcwd(), "Testinstanzen", "Instance01_m3_n20.json"))
evalutation = EvaluationLogic(data)

sol = Solution([2, 2, -1, 1, 1, 1, 2, 0, -1, 2, 2, -1, 0, 1, 1, -1, 2, 2, -1, 1])
evalutation.calcProfit(sol)
sol.printSolution()
print(sol)
sol.to_json("testloesung.json", force = True, includeEarnings = True, includePenalty = True)
```

```
Knapsack 0: [7, 12]
Knapsack 1: [3, 4, 5, 13, 14, 19]
Knapsack 2: [0, 1, 6, 9, 10, 16, 17]
Solution(allocation=[2, 2, -1, 1, 1, 1, 2, 0, -1, 2, 2, -1, 0, 1, 1, -1, 2, 2, -1, 1], penalty=136, earnings=2
25, profit=89)
```

Um eine erste konstruktive Lösung zu ermitteln, wird versucht die Knapsacks nacheinander mit Items zu füllen, bis ein Knapsack voll ist. Dann wird der nächste Knapsack gefüllt, usw. Um die Lösung gleich ein bisschen zu verbessern, werden die Items nach Profit/Weight absteigend sortiert, es werden also bevorzugt kleine und wertvolle Items mitgenommen. Weiterhin werden die Knapsacks nach Penalty absteigend sortiert, damit der erste Knapsack möglichst voll wird und wenig Strafkosten generiert (am Anfang werden ja insbesondere kleine Items mitgenommen, damit lässt sich der Platz gut füllen).

Die so gefundene Lösung wird evaluiert und in den SolutionPool hinzugefügt.

Die Evalutation sieht so aus, dass ich zuerst eine Liste mit dem aktuellen Gewicht eines jeden Knapsacks initialisiere (am Anfang besteht diese logischerweise nur aus Nullen). Dann itereiere ich durch die Lösungsrepräsentation und füge das Gewicht eines jeden Items zum aktuellen Gewicht des Knapsacks hinzu und addiere auch den Profit des Items. Ist das abgeschlossen, so folgt die Zulässugkeitsprüfung. Dafür schaue ich mir meine Gewichte in den Knapsacks an und vergleiche, ob diese über dem maximal zulässigen Gewicht liegen. Ist das der Fall, so wird `valid = False` gesetzt. In dem selben Schritt können auch gleich die Penalties berechnet werden. Der tatsächliche Profit der Lösung ist dann die Summe aller Item-Profits minus die Summe der Penalties.

In [3]:
```
from cls.ConstructiveHeuristic import ConstructiveHeuristics
from cls.SolutionPool import SolutionPool

pool = SolutionPool()
heuristik = ConstructiveHeuristics(EvaluationLogic(data), pool)
heuristik.Run(data, "greedy")
print(pool)
```

```
Generating an initial solution according to greedy.
SolutionPool(Solutions={Solution(allocation=[2, 2, -1, 1, -1, 1, 2, 0, 1, 2, 2, -1, 0, 1, 1, -1, 2, 2, -1, 1],
penalty=159, earnings=241, profit=82)})
```

Für die Nachbarschaften musste ich dann immer nur Überprüfen, ob eine Lösung gültig ist oder nicht, deswegen gibt `EvaluationLogic.calcProfit()` auch immer einen Boolean zurück, ob die Lösung gültig ist. Ist eine Lösung nicht gültig, so wird diese nicht den MoveSolutions hinzugefügt. Es kann natürlich sein, dass eine Nachbarschaft mal gar keine gültige Lösung erzeugen kann, aber die nächste Nachbarschaft aus `BaseNeighborhood.MoveSolutions` die beste Lösung haben will. Im Fall, dass dieser leer ist, wird die beste Lösung aus dem SolutionPool zurückgegeben, indem ja mindestens immer die gültige konstruktive Lösung liegt.

Es sind folgende Nachbarschaften implementiert:

- Swap: Tauscht zwei Items
- Insertion: Fügt ein Item mit einer bestimmten ID in einen Knapsack mit einer bestimmten ID ein
- BlockMoveK3: (kommt aus dem letzten Semester)
- TwoEdgeExchange: tauscht einige Items gleichzeitig aus

Nach meinen bisherigen Erfahrungen erzeugen BlockMoveK3 und TwoEdgeExchange nur selten gültige Lösungen.

In [4]:
```
from cls.Neighborhood import SwapNeighborhood

swapN = SwapNeighborhood(data, pool.GetHighestProfitSolution().allocation, EvaluationLogic(data), pool)
swapN.DiscoverMoves()
swapN.EvaluateMoves("BestImprovement")
print(swapN.MakeBestMove())
```

```
Solution(allocation=[2, 2, -1, 1, 1, 1, 2, 0, -1, 2, 2, -1, 0, 1, 1, -1, 2, 2, -1, 1], penalty=136, earnings=2
25, profit=89)
```

Die Tabu Search ist ein ImprovementAlgorithm, der eine gewissen Anzahl an Iterationen (auch Zeit) durchlaufen kann. Es wird zu einer Startlösung mittels IterativeImprovement eine Nachbarschaft aufgebaut und die beste Lösung in dieser Nachbarschaft rausgesucht. Ist diese Nachbarschaft allerdings in der Tabu Liste, so wird diese nicht angenommen, es sei denn, sie erfüllt ein Aspirationskriterium (besser als aktuell beste Lösung). Ich habe mich für ein langfristiges Gedächtnis entschieden, das war am leichtesten zu implmentieren, da nie Elemente aus der Tabu Liste entfernt werden müssen.

Ich habe später festgestellt, dass das langfristige Gedächtnis nicht die beste Idee ist und habe deswegen eine kleine Abwandlung implementiert: Neben der besten Lösung von IterativeImprovement hole ich mir auch andere gute Lösungen. Sollte meine beste Lösung in der Tabu-Liste sein, so schaue ich nach, ob die zweitbeste Lösung in der Tabu-Liste ist (falls ja, dann die drittbeste usw.). Diese Lösung kommt dann auf die Tabu-Liste und der Prozess startet von vorn mit der zweitbesten (drittbesten, etc.) Lösung als Startlösung. Damit verhindere ich das Cycling.

Man kann bei der Tabu Search einen Paramter für die Dauer einstellen (`maxSeconds`), wie lange diese laufen soll. Oder man stellt ein, wie viele Iterationen die Tabu Search laufen soll (`maxIterations`). Beide Parameter müssen jetzt werden und sobald eines der beiden Stoppkriterien erreicht ist, hört die Tabu Search auf.

In [5]:
```
from cls.Solver import Solver
from cls.ImprovementAlgorithm import TabuSearch

solver = Solver(data, 42)
tabu = TabuSearch(data, maxSeconds = 10, maxIterations = 999, neighborhoodEvaluationStrategy = "BestImprovemer
bestSol = solver.RunLocalSearch("greedy", tabu)
```

```
Generating an initial solution according to greedy.
Constructive solution found.
Solution(allocation=[2, 2, -1, 1, -1, 1, 2, 0, 1, 2, 2, -1, 0, 1, 1, -1, 2, 2, -1, 1], penalty=159, earnings=2
41, profit=82)
Best found Solution.
Solution(allocation=[2, 2, -1, 1, 1, 1, 2, 0, -1, 2, 2, -1, 0, 1, 1, -1, 2, 2, -1, 1], penalty=136, earnings=2
25, profit=89)
```

Auswertung

Die folgenden Ergebnisse kommen von 30 Iterationen (und 10 Minuten Dauer) der Tabu Search. Die eingesetzten Verfahren sind alle deterministisch, damit ist die Lösung unabhängig vom Seed.

Nicht von allen Testinstanzen habe ich die optimalen Lösungen und die Testinstanzen *Instance10_m25_n500.json*, *Instance16_m60_n240.json* und *Instance25_m75_n225.json* dauern zu lange in der Berechnung (ich habe es nach 10 Minuten abgebrochen)

Testinstanz	Zeit in Sekunden	konstruktive Lösung	gefundene Lösung	prozentuale Verbesserung	optimale Lösung	Optimalität der gefundenen Lösung
Instance05_m10_n60.json	28.616091	2631	16785	537.97 %	26052.0	64.43 %
Instance03_m5_n40.json	4.50168	-1208	265	555.85 %	785.0	33.76 %
Instance00_m2_n20.json	0.834147	-429	184	333.15 %	218.0	84.4 %
Instance04_m10_n40.json	6.541545	-37901	2600	1557.73 %	15805.0	16.45 %
Instance13_m30_n60.json	20.115957	-17813	-17099	4.18 %		
Instance08_m15_n75.json	74.416179	-6982	18214	138.33 %	31957.0	57.0 %
Instance5_m20_n60.json	24.862974	16925	20481	21.01 %		
Instance07_m5_n75.json	28.772266	-357	729	148.97 %	879.0	82.94 %
Instance1_m10_n40.json	4.517878	-1169	2679	143.64 %		
Instance7_m9_n100.json	88.219469	9060	15283	68.69 %		
Instance01_m3_n20.json	0.919188	82	89	8.54 %	238.0	37.39 %
Instance09_m10_n100.json	166.298572	15327	36619	138.92 %	42502.0	86.16 %
Instance0_m20_n20.json	1.109725	-146	1734	108.42 %		
Instance02_m3_n40.json	4.195113	-815	220	470.45 %	270.0	81.48 %
Instance06_m10_n60.json	25.812224	-1136	13395	108.48 %	26436.0	50.67 %
Instance8_m12_n48.json	10.552388	6611	11048	67.12 %		
Instance6_m10_n100.json	91.94361	7778	16243	108.83 %		

Die Berechnung für die prozentuale Verbesserung ist nicht ganz offensichtlich:

$$\text{prozentuale Verbesserung} = \begin{cases} \left| \frac{\text{gefundene Lösung} - \text{konstruktive Lösung}}{\text{konstruktive Lösung}} \right| & \text{konstruktive Lösung} > 0 \\ \left| \frac{\text{gefundene Lösung} - \text{konstruktive Lösung}}{\text{gefundene Lösung}} \right| & \text{konstruktive Lösung} \leq 0 \end{cases} \quad (1)$$

Die Tabu Search erreicht also starke Verbesserungen gegenüber der konstruktiven Heuristik. Interessanterweise erreicht die Tabu Search die oben angegebenen Ergebnisse schon nach wenigen Iterationen, Verdreifacht man z.B. die Anzahl der Iterationen (und setzt das Zeitlimit) sehr hoch, so lässt sich keine Verbesserung erzielen. Das liegt vermutlich an der Tabu Liste, da diese nicht gelehrt wird, sondern immer länger wird und damit neue, möglicherweise gute, aber nicht sehr gute Lösungen, in der nächsten Iteration weiter verbessert werden können.

Besonders hervorzuheben sind hier die Testinstanzen *Instance13_m30_n60.json* und *Instance02_m3_n40.json*. Bei ersterer ist der Profit negativ; bei zweiterer ist die Penalty 0, das heißt der Platz in den Knapsacks wurde perfekt ausgenutzt.

Schaut man sich die Daten an, so scheint es, als wird meine gefundene Lösung "optimaler", wenn es mehr Items gibt. Zwischen der Anzahl an Items und der "Optimalität der Lösungg", gemessen in Prozent, gibt es eine Korrelation von 44%, also einen mittelstarken Zusammenhang.

Die kleine Abwandlung in der Tabu Search hat am Ende doch nicht so viel gebracht wie erhofft. Einige Lösungen sind besser, einige Lösungen schlechter (und zwar unter anderem die, die wir in unserer Präsentation als Vergleich nutzen 😅). In Summe hält sich das aber die Waage, weswegen ich die Änderung nicht rückgängig mache.