Knapsack problem je NP-ťažká úloha kombinatorickej optimalizácie. Majúc batoh s kapacitou M a počet predmetov N, z ktorých každý má svoju hmotnosť m_i a hodnotu v_i, cieľom je nájsť takú podmnožinu týchto predmetov, ktorá by neprekračovala kapacitu batohu a zároveň by optimalizovala celkový profit.

1 Teoretický úvod

Úlohu sme mali riešiť pomocou troch rôznych heuristík. Za tie sme si vybrali greedy algoritmus, metódu náhodného spádu a simulované žíhanie. Tieto sú rozobrané v podsekciách nižšie.

Greedy Approach

Algoritmom vyvinutým špeciálne pre riešenie problémov ako je problém batohu bol vyvinutý tzv. greedy algoritmus [1]. Za greedy sú považované tie algoritmy, ktoré v každom kroku volia lokálne optimálne riešenie. Pri riešení problému batohu sa pomocou greedy algoritmu postupuje nasledujúcim spôsobom:

1. spočíta sa jednotková hodnota každej z dostupných predmetov, tj. v_i/m_i a tieto sa zoradia zostupne,
2. predmet s najvyššou jednotkovou hodnotou sa následne pridáva do batohu, dokým ďalšie pridanie nespôsobí pretečenie kapacity batohu.

Za predpokladu, že máme k dispozícii neobmedzené množstvo každého predmetu a v_{max} je maximálna hodnota, ktorú je batoh schopný pri daných predmetoch poňať, potom greedy algoritmus zaručuje dosiahnutie hodnoty aspoň $v_{max}/2$.

Toto však nemusí platiť pre problém s obmedzeným počtom predmetov. V tomto prípade sa dá algoritmus modifikovať nasledovne. Okrem pôvodného greedy riešenia, ktoré zahŕňa predmety do dosiahnutia kapacity, sa vytvorí druhé riešenie, kde je zahrnutá prvá vec, ktorá sa do batohu nevošla. Tieto riešenia sa porovnajú a vyberie sa to lepšie z nich.

Random Descent

Metóda náhodného spádu [3] je iná optimalizačná metóda, ktorá sa snaží nájsť minimum danej funkcie iteratívnym spôsobom. Pre hladkú funkciu f má metóda nasledovný tvar. Najskôr sa zvolí počiatočný bod $x_0 \in \mathbf{R}^n$. Pre zvolený počet iterácií sa potom vykonávajú nesledujúce príkazy:

1. podľa rovnomerného rozdelenia sa náhodne zvolí súradnica $i \in \{1, 2, ..., n\}$,
2. x sa aktualizuje na základe $x^{(i)} = x^{(i)} - \frac{1}{L_i}\nabla_i f(x_0)$.

Tu L_i predstavuje Lipschitzovskú konštantu a ∇_i symbolizuje parciálnu deriváciu f v premennej $x^{(i)}$.

Simulated Annealing

Poslednou zvolenou metódou bolo simulované žíhanie [2], ktorého názov vychádza z podobnosti žíhania v metalurgii. Jedná sa o pravdepodobnostnú metódu aproximujúcu globálne optimálne riešenie. Často sa používa práve pre diskrétne úlohy, napríklad úloha obchodného cestujúceho, a je vhodná v prípade, kedy preferujeme nájsť aproximálne globálne optimum pred presnými lokálnymi optimami.

Podstata metódy spočíva v tom, že v každom kroku heuristika uvažuje súčasný stav systému s a susediaci stav systému $\hat{s}$. To, či sa systém posunie zo stavu s do stavu $\hat{s}$, sa určuje na základe pravdepodobností. Systém je pod vplyvom týchto pravdepodobností preskúmať stavy s nižšou energiou, než sú súčasné stavy, čím sa efektívne vyhýba uviaznutiu v lokálnom optime.

Na počiatku sa zvolí počiatočný stav s_0. Ďalej je potreba zvoliť parametre žíhania, konkrétne počiatočnú teplotu T_0, cooling rate r_C a počet iterácií k_{max}. Postup je nasledovný:

1. teplota klesá podľa nastaveného cooling rate ako $T = T * r_C$,
2. náhodne sa vyberie nový (susediaci) stav,
3. ak je nový stav energeticky výhodnejší než pôvodný stav, nahradí ho na základe náhodne vybraného čísla z intervalu (0, 1), ktoré slúži ako pravdepodobnosť.

Algoritmus sa ukončí po dosiahnutí maximálneho počtu korkov k_{max}.

2 Heuristické riešenie

V tejto sekcii implementujeme vyššie popísané metódy a porovnáme ich výsledky. Všetky spomínané metódy boli implementované formou vlastných funkcií. Script `main` definuje predmety, ich hodnoty, váhy a kapacitu batohu. Ďalej sú tu taktiež uvedené všetky parametre pre simulované žíhanie.

Váhy predmetov m_i a ich hodnoty v_i spolu s kapacitou batohu boli definované nasledovne,

```
weights = [2, 3, 4, 5, 6];
values = [10, 20, 30, 40, 50];
capacity = 10;
```

Je vidno, že maximálnu hodnotu je možné dosiahnuť zabalením predmetov s váhou 6 a 4, čím obdržíme profit 80. Parametre pre simulované žíhanie, tj. počiatočná teplota T_0, cooling rate r_C a maximálny počet iterácií k_{max}, sú potom v uvedenom poradí definované ako

```
initialTemperature = 100;
coolingRate = 0.95;
numIterations = 1000;
```

Nakoľko algoritmy náhodného zostupu a simulovaného žíhania vyžadujú nejaké počiatočné riešenie, je nutné toto vygenerovať tak, aby spĺňalo obmedzenie kapacity. K tomu slúži nasledujúca funkcia.

```
function currentSolution = initKnapsack(weights, capacity)
while true
        currentSolution = randi([0, 1], size(weights));
        if weights*currentSolution' > capacity
        else
                break;
        end
end
```

Greedy Approach

V praxi vybral tento algoritmus vždy riešenie `[0,0,0,0,1]` zodpovedajúce predmetu s váhou 6 a hodnotou 50. Aj v prípade obmedzenej zásoby jednotlivých predmetov tak bola dosiahnutá celková hodnota väčšia ako $v_{max}/2$, ktorá sa v tomto prípade rovná 40.

```
function [solution, value] = greedyKnapsack(weights, values, capacity)
numItems = length(weights);
valuePerWeight = values ./ weights;
[~, sortedIndices] = sort(valuePerWeight, 'descend');
solution = zeros(1, numItems);
remainingCapacity = capacity;
for i = 1:numItems
        if weights(sortedIndices(i)) <= remainingCapacity
```

```
                solution(sortedIndices(i)) = 1;
                value = values*solution';
                remainingCapacity = remainingCapacity - weights(sortedIndices(i));
        else
                break;
        end
end
end
```

Random Descent

V prípade metódy náhodného spádu sa využíva myšlienka náhodného výberu súradnice spomenutá v teoretickej časti na príklade funkcií. Táto súradnica označuje predmet, ktorý bude pridaný alebo odobraný z batohu v závislosti na tom, či sa v ňom už nachádza alebo nie. Následne sa skúma celková hodnota tohto nového riešenia. Pokiaľ nedôjde k prekročeniu kapacity a zároveň sa zvýši hodnota v batohu, tak pôvodné riešenie nahradíme starým. Takto sa pokračuje až dokým cyklus neprebehne vštkých k_{max} krokov.

```
function [currentSolution, currentValue] = randomKnapsack(weights, values,...
capacity, initSolution)
currentValue = values*initSolution';
nextSolution = initSolution;

while true
        currentSolution = nextSolution;
        currentValue = values*currentSolution';
        idx = randi(length(initSolution));
        nextSolution(idx) = ~nextSolution(idx);
        neighborValue = values*nextSolution';
        neighborWeight = weights*nextSolution';

        if neighborWeight <= capacity && neighborValue > currentValue
        else
                break;
        end
end
```

Simulated Annealing

Simulované žíhanie má podobný priebeh ako metódan náhodného zostupu. Jediný rozdiel spočíva v tom, že podmienka neprekročenia kapacity a zvýšenia hodnoty je ešte rozšírená o výraz `rand < exp((neighborValue - currentValue) / currentTemperature)`. Ako bolo spomínané vyššie, tá slúži k lepšiemu preskúmaniu priestoru riešení a vylepšuje tak schopnosť metódy uniknúť z lokálnych extrémov.

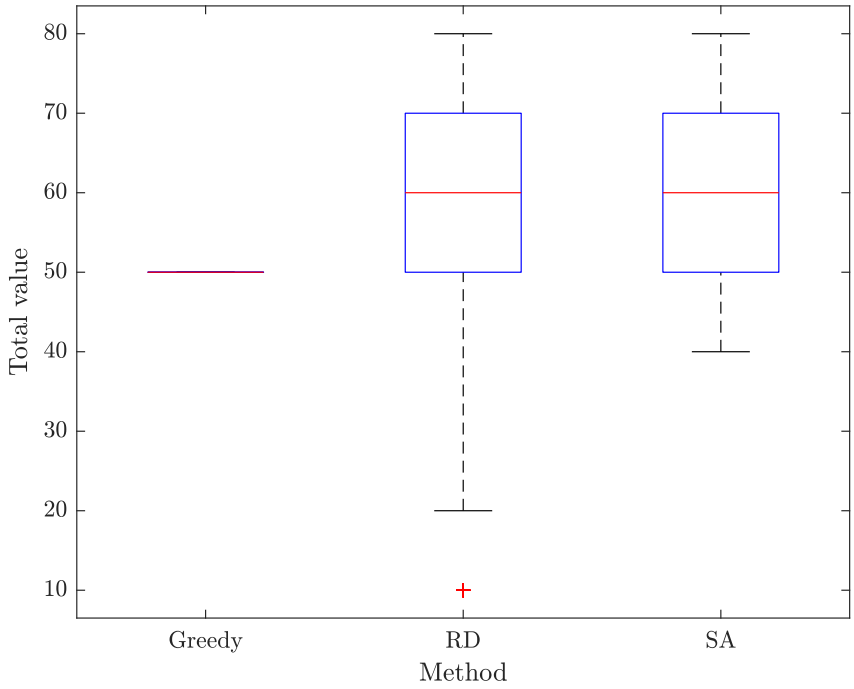
```
function [currentSolution, currentValue] = annealingKnapsack(weights, values,...
capacity, initialTemperature, numIterations, coolingRate, initSolution)
currentValue = values*initSolution';
currentTemperature = initialTemperature;

for i = 1:numIterations
        neighborSolution = initSolution;
        idx = randi(length(initSolution));
        neighborSolution(idx) = ~neighborSolution(idx);         % neighbour selection
        neighborValue = values*neighborSolution';
        neighborWeight = weights*neighborSolution';
```

```
        if neighborWeight <= capacity && (neighborValue > currentValue ||...
                rand < exp((neighborValue - currentValue) / currentTemperature))
                currentSolution = neighborSolution;
                currentValue = neighborValue;
        else
                currentSolution = initSolution;
        end
currentTemperature = currentTemperature*coolingRate;
end
```

Vyhodnotenie

Všetky tri metódy sme nechali prebehnúť cez 1000 cyklov s rôznymi iniciaizáciami počiatočného riešenia. Výsledky sme graficky vykreslili pomocou boxplotu na obr. 1. Z obrázku je vidieť, že najväčší rozptyl má metóda



Obr. 1: Rozptyl výsledkov pre jednotlivé metódy.

náhodného spádu (RD), ktorá u niektorých inicializácií dospela iba k celkovej hodnote 20. Oproti greedy algoritmu ale spolu so simulovaným žíhaním (SA) dosiahli aj globálne optimálneho riešenia, ktorým je hodnota 80.

Naproti tomu greedy algoritmus dospel v každom z 1000 cyklov k hodnote 50, ktorá zodpovedá predmetu s najvyššou jednotkovou hodnotou. Táto metóda nezávisela na počiatočnej inicializácii a nemala v sebe implementovaný žiadny náhodný prvok, preto je rozptyl výsledkov v jej prípade nulový. Nájdené optimálne hodnoty pre rôzne počiatočné riešenia spriemerované cez všetkých 1000 cyklov sú k videniu nižšie.

```
Average total value using greedy heuristic: 50
Average total value using random descent: 57.71
Average total value using simulated annealing: 61
```

3 Záver

Po priemerovaní cez všetky behy s rôznymi počiatočnými riešeniami dosiahlo najlepšie výsledky simulované žíhanie, konkrétne hodnotu 61. Iba o málo horšie dopadla metóda náhodného spádu, ktorou sme obdržali zaokrúhlene hodnotu 58. Mierny pokles výkonu mohol byť spôsobený aj väčším rozptylom metódy, ktorý je viditeľný na obr. 1. Nízke hodnoty okolo 20 mohli priemer stiahnuť o málo nižšie. Oproti tomu najnižšie hodnoty obdržané simulovaným žíhaním sa pohybujú okolo 40. U oboch skúmaných metód ale 75 % výsledkov spadlo nad hodnotu 50, čo je jediná hodnota obdržaná použitím greedy algoritmu.

Rozhodovaním medzi greedy algoritmom, metódou náhodného spádu a simulovaným žíhaním by sa teda mohlo zdať najvýhodnejšie využiť jednu z posledných dvoch spomínaných heuristík. Výsledná implementácia však záleží na type riešenej úlohy. Ako bolo rozoberané vyššie, tieto dve metódy majú zároveň podstatne väčší rozptyl výsledkov, a tak je možné dosiahnuť aj hodnoty batohu nižšie (a v prípade náhodného spádu aj podstatne nižšie) než greedy algoritmus. V úlohách, kedy výslednému používateľovi ide hlavne o navýšenie najnižšej možnej hodnoty než o maximalizáciu možného zisku, tak môže byť výhodnejšie použiť greedy algoritmus než simulované žíhanie alebo náhodný spád.

Zdroje

[1] G. B. Dantzig. Discrete-variable extremum problems. *Operations Research*, 5(2):266–288, 1957.

[2] S. Kirkpatrick, C. D. Gelatt Jr, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):471–480, 1983.

[3] P. Richtárik and M. Takáč. Iteration complexity of randomized block-coordinate descent methods for minimizing a composite function. *Mathematical Programming, Series A*, 22(1-2):1–38, 2011.