Multiple Category Knapsack Problem

Bc. Zdeňka Kolářová

Seminární práce
2023

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

OBSAH

1 DEFINOVÁNÍ PROBLÉMU A GENEROVÁNÍ PŘEDMĚTŮ

Pro zpracování jsem si vybrala problém batohu s vícenásobnou volbou (Multiple-choice knapsack problém, MKCP). Pro všechny dimenze problému jsou vygenerovány 3 předměty z každé kategorie. Každý předmět má přiřazenou cenu (price) a váhu (weight) v podobě celočíselné hodnoty od 1 do 50. Kapacita batohu je 100 při 5 kategoriích, 200 při 6-10 kategoriích a 300 při větším počtu kategorií.

CATEGORY	ID IN CATEGORY	PRICE	WEIGHT
1	1	13	39
1	2	30	3
1	3	8	16
2	1	41	21
2	2	25	38
2	3	21	37
3	1	20	33
3	2	43	45
3	3	3	41
4	1	35	47
4	2	34	34
4	3	4	25
5	1	34	34
5	2	13	21
5	3	44	16
TOTAL PRICE	**368**	**TOTAL WEIGHT**	**450**

Tabulka 1 – Vygenerované předměty při dimenzi 5

Předměty jsem nejprve generovala do .csv souborů a při pozdějších spouštěních už pouze načítala jejich obsah. Při nižším počtu kategorií jsem žádná speciální omezení implementovat nemusela, ale pokud se při zvyšování dimenze problému dostatečně nezvyšuje také kapacita batohu, narůstá pravděpodobnost, že vygenerovaná množina předmětů nebude umožňovat žádný výběr, který by kapacitnímu omezení vyhovoval.

```
def generate_item_catalogue(dir_path, filename,
category_count, items_per_category):
    low_border = 1
    up_border = 50
    price_sum = 0
    weight_sum = 0
    full_file_path = f"{dir_path}/{filename}"
    file = open(full_file_path, "a")
    file.write(Item.header())
    for category in range(1, category_count + 1):
```

```
        for item_id in range(1, items_per_category + 1):
            price = random.randint(low_border, up_border)
            weight = random.randint(low_border, up_border)
            file.write(Item.data_row(Item(category, item_id,
price, weight)))
            price_sum += price
            weight_sum += weight
    file.write(Item.footer(price_sum, weight_sum))
```

Ukázka kódu 1 – Generování množiny předmětů bez omezení

Váhový limit 300, který jsme měli použít od dimenze 11 výše, se sice např. při dimenzi 21 podařilo dodržet hned s první vygenerovanou množinou předmětů, ale při dimenzi bylo pokusů potřeba více.

```
viability_check = 301
while viability_check > 300:
    viability_check = 0
    de.generate_item_catalogue(dirname, catalogue_filename,
category_count, items_per_category)
    items = de.load_from_item_catalogue(full_catalogue_path)
    for i in range(category_count):
        start_index = i * items_per_category
        smallest_weight_in_category =
items[start_index].weight
        for j in range(items_per_category - 1):
            current_weight = items[start_index + 1 +
j].weight
            if current_weight < smallest_weight_in_category:
                smallest_weight_in_category = current_weight
        viability_check += smallest_weight_in_category
```

Ukázka kódu 2 – Kontrola, že množina předmětů bude mít platné řešení

Jako účelovou funkci (Cost Function, CF) jsem zvolila prostý součet cen vybraných předmětů, takže 2 řešení se stejným součtem hodnot, ale různými hmotnostmi, si budou rovna. Alternativním řešením by mohlo být preferování lehčích sad předmětů, kterého by při dodržení celočíselnosti hodnot šlo docílit např. přičítáním invertovaného součtu hodnot předmětů.

použitá Cost Function:

$$CF = \Sigma(p_i)$$

alternativní Cost Function:

$$CF = \Sigma(p_i) + \frac{1}{\Sigma w_j}$$

2 BRUTEFORCE

Čas k nalezení řešení hrubou silou při zvyšování dimenze zvyšuje úměrně počtu předmětů v kategorii. Při 3 předmětech na kategorii dokáže můj notebook prozkoumat všechna možná řešení za méně než hodinu do dimenze 19, přičemž počet všech kombinací předmětů již překračuje miliardu.

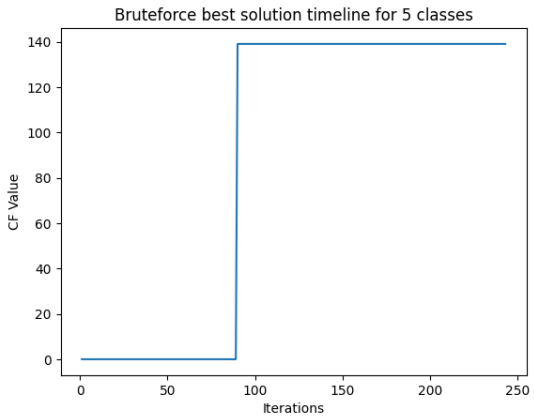
Jedno možné zefektivnění řešení hrubou silou, o kterém jsem přemýšlela, ale neimplementovala, je přeskakování řešení obsahujících neplatné kombinace. Např. pokud při 5 dimenzích dojde k překročení kapacity batohu už po započtení prvních 3 předmětů z dané kombinace, přes čtvrtou a pátou kategorii už není nutné iterovat a je možné rovnou přejít k dalšímu předmětu ve třetí kategorii. Tahle optimalizace by mohla vést k významným časovým úsporám obzvlášť ve vyšších dimenzích.

Dimension	Items per category	Total solutions	Time to bruteforce
5	3	243	<1 ms
6	3	729	2 ms
7	3	2187	5 ms
8	3	6561	14 ms
9	3	19683	48 ms
10	3	59049	129 ms
11	3	177147	460 ms
12	3	531441	1,5 s
13	3	1594323	4,5 s
14	3	4782969	14,5 s
15	3	14348907	42 s
16	3	43046721	2 min 11 s
17	3	129140163	7 min
18	3	387420489	20 min
19	3	1162261467	57 min
20	3	3486784401	3 h 11 min
21	3	10460353203	9 h 11 min

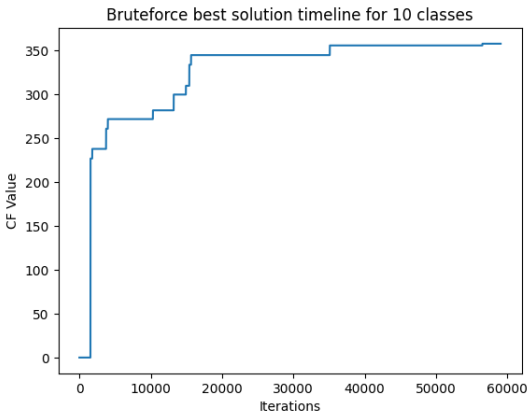
Tabulka 2 – Časová náročnost bruteforce MCKP při 3 předmětech na kategorii

Graf průběžné hodnoty nejlepšího nalezeného řešení v případě bruteforce MKCP závisí čistě na uspořádání vygenerované množiny předmětů. V případě dimenze 5 u mě došlo k zajímavé situaci, kdy hned první nalezené vyhovující řešení bylo tím nejlepším.

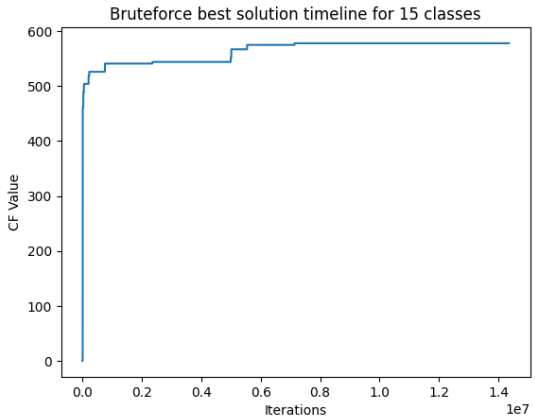
Nejvyšší dimenze, pro kterou ještě mohu vygenerovat graf vývoje bruteforce řešení, kde 1 bod linie = 1 řešení, je dimenze 17, kde počet řešení přesahuje 129 milionů. Tvar grafu s dimenzí 17 (obrázek 3) reprezentuje moje získané výsledky mnohem lépe.



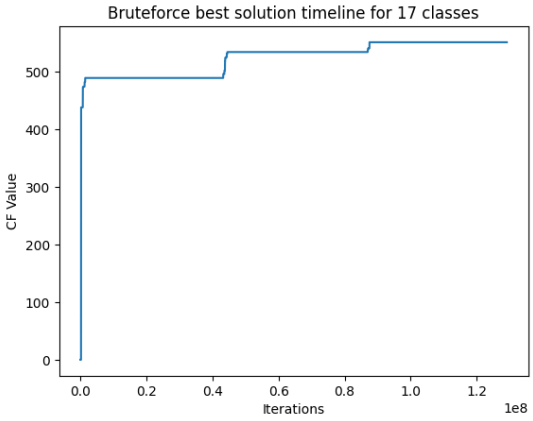
Obrázek 1 – Vývoj nejlepšího nalezeného řešení použitím bruteforce při dimenzi 5



Obrázek 2 – Vývoj nejlepšího nalezeného řešení použitím bruteforce při dimenzi 10



Obrázek 3 – Vývoj nejlepšího nalezeného řešení použitím bruteforce při dimenzi 15



Obrázek 4 – Vývoj nejlepšího nalezeného řešení použitím bruteforce při dimenzi 17

3 SIMULOVANÉ ŽÍHÁNÍ

Simulované žíhání se ukázalo být velmi efektivním způsobem řešení problému tam, kde bruteforce prochází všechny možné kombinace déle než minutu. Použila jsem konfiguraci, k níž jsem došla v benchmarkové úloze s drobnými modifikacemi.

Parametr	**Hodnota**	**Změna oproti benchmark úloze**
hlavních iterací (A)	$MIN\left(\frac{celkový\ počet\ řešení}{dimenze \cdot předmětů\ v\ kategorii}, 10\ 000\right)$	nižší počet hlavních iterací u dimenzí s menšími celkovými počty řešení
hlavních iterací (B)	$MIN\left(\frac{celkový\ počet\ řešení}{dimenze \cdot předmětů\ v\ kategorii}, 100\right)$	nižší počet hlavních iterací u dimenzí s menšími celkovými počty řešení
iterací metropolisu	dimenze	s dimenzí problému zvyšuju počet iterací metropolisu
počáteční teplota	1 200	beze změny
minimální teplota	0,1	beze změny
chladicí dekrement	0,94	beze změny

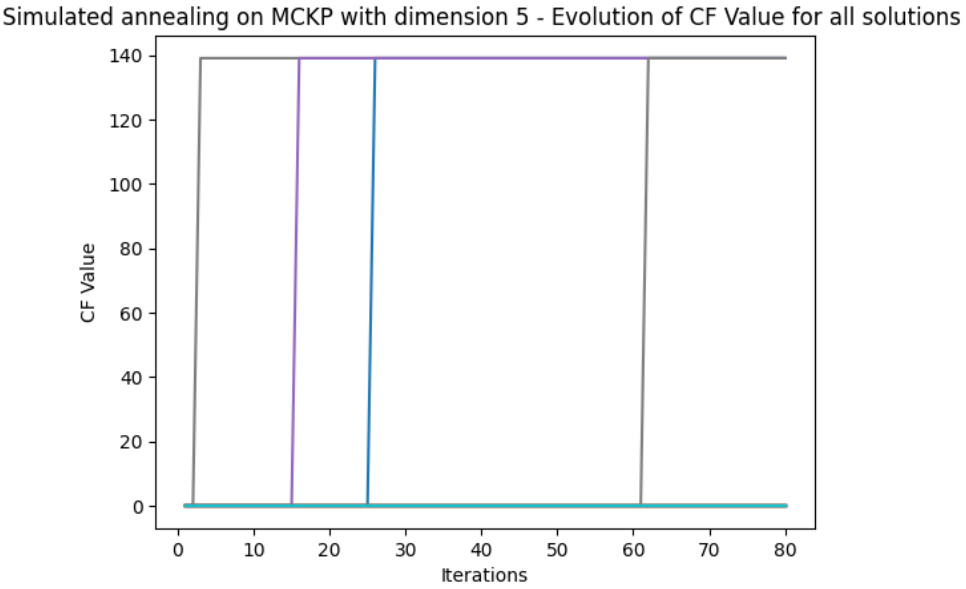
Tabulka 3 – Parametry použité pro simulované žíhání

Moje první konfigurace simulovaného žíhání (A) začala být rychlejší než bruteforce až při dimenzi 15, a tak jsem přistoupila ke konfiguraci B.

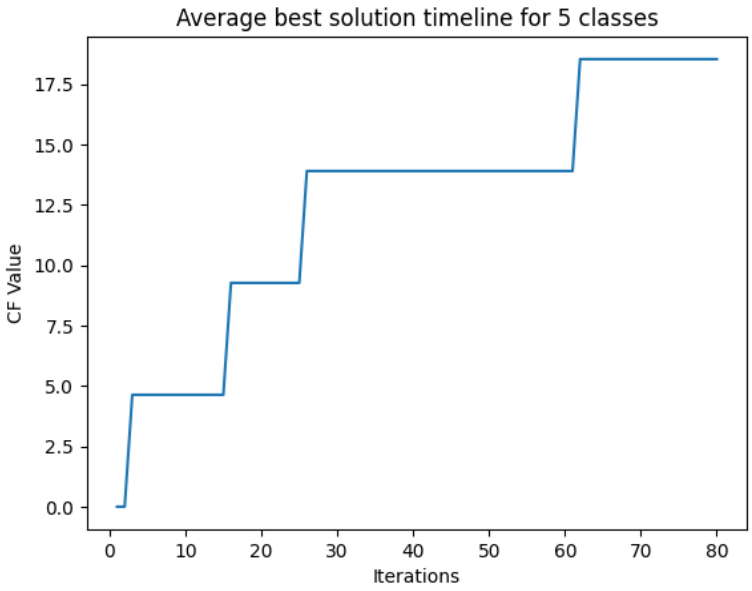
D	Best possible solution	Configuration A				Configuration B			
		Total SA itera-tions	30 SA runs time	Best SA result	Best SA to best BF	Total SA itera-tions	30 SA runs time	Best SA result	Best SA to best BF
5	139	80	41 ms	139	100,00%				
6	150	240	102 ms	150	100,00%				
7	243	728	201 ms	243	100,00%	700	151 ms	243	100,00%
8	356	2184	487 ms	356	100,00%	800	179 ms	356	100,00%
9	305	6561	1,4 s	305	100,00%	900	212 ms	305	100,00%
10	358	19680	4,4 s	358	100,00%	1000	243 ms	356	99,44%
11	418	59048	13,2 s	418	100,00%	1100	288 ms	404	96,65%
12	435	120000	28 s	435	100,00%	1200	311 ms	430	98,85%
13	458	130000	29 S	457	99,78%	1300	386 ms	435	94,98%
14	526	140000	38 s	526	100,00%	1400	411 ms	480	91,25%
15	578	150000	41 s	575	99,48%	1500	450 ms	530	91,70%
16	498	160000	43 s	485	97,39%	1600	503 ms	399	80,12%
17	551	170000	48 s	540	98,00%	1700	500 ms	509	92,38%
18	657	180000	50 s	645	98,17%	1800	577 ms	607	92,39%
19	555	190000	51 s	500	90,09%	1900	604 ms	X	-
20	676	200000	56 s	610	90,24%	2000	614 ms	X	-
21	580	210000	58 s	X	-	2100	645 ms	X	-

Tabulka 4 – Výsledky simulovaného žíhání

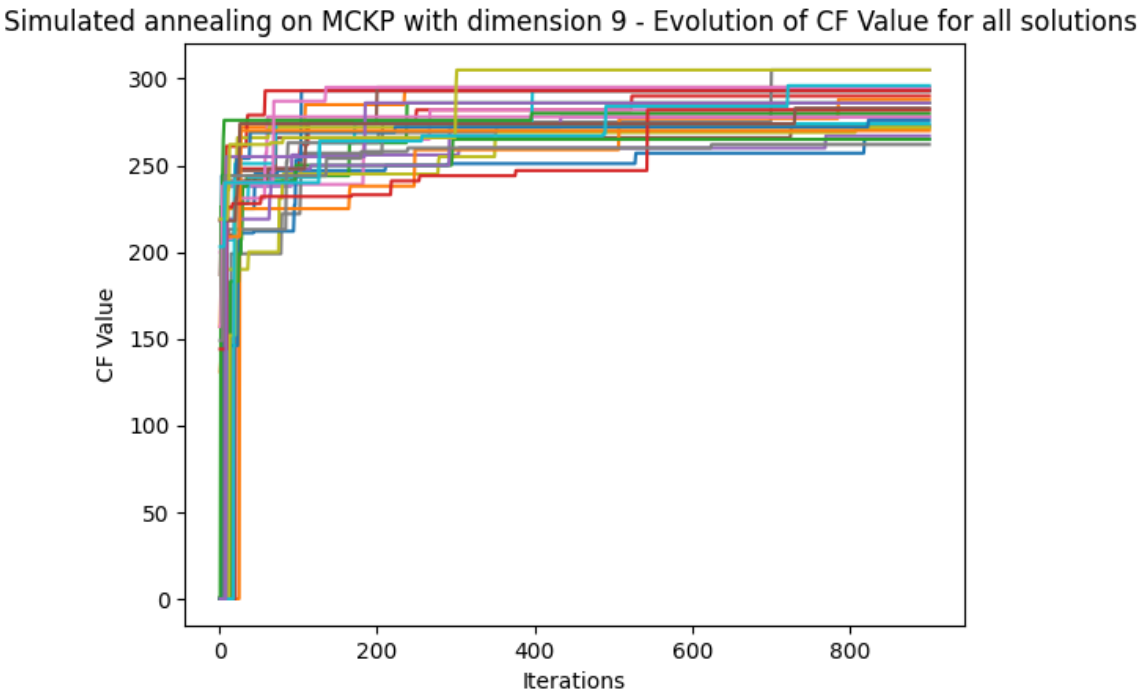
Konfigurace B provádí jen velmi malý počet iterací, což ale v nízkých dimenzích k nalezení přiměřeně dobrého řešení naprosto dostačuje. Se vzrůstající dimenzí by naopak bylo dobré počet iterací ještě zvýšit.



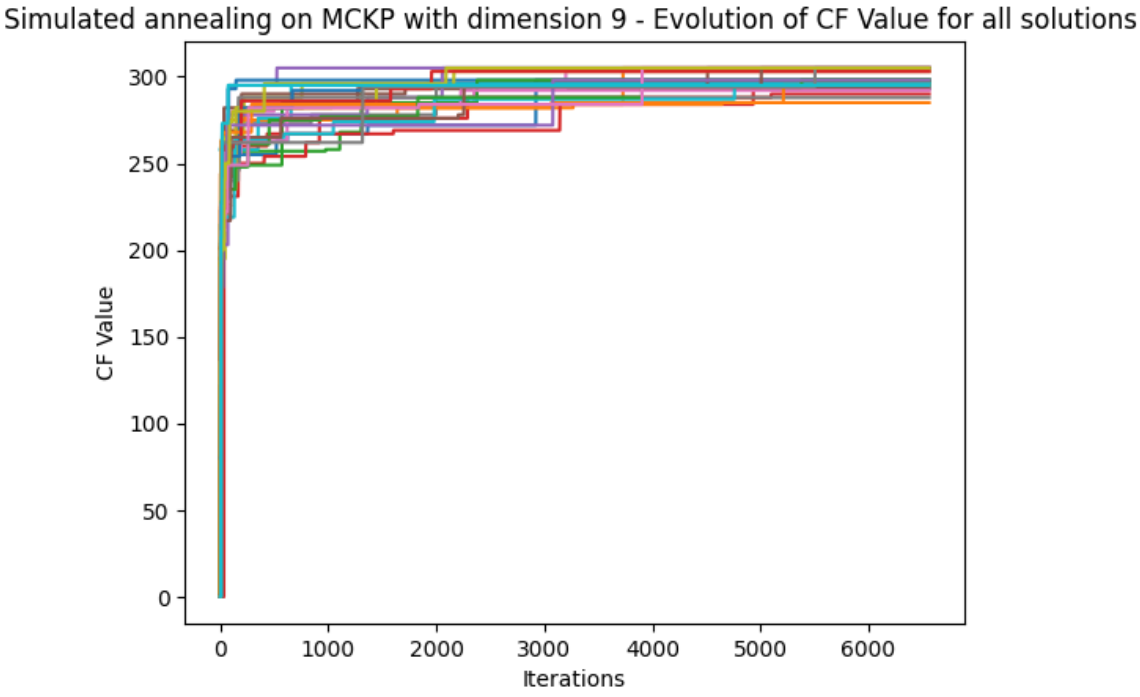
Obrázek 5 – 30 běhů SA, dimenze 5, konfigurace A



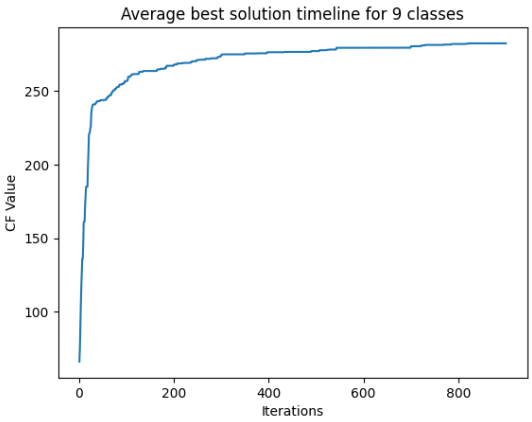
Obrázek 6 – Průměr nejlepších řešení ze 30 běhů SA, dimenze 5, konfigurace A



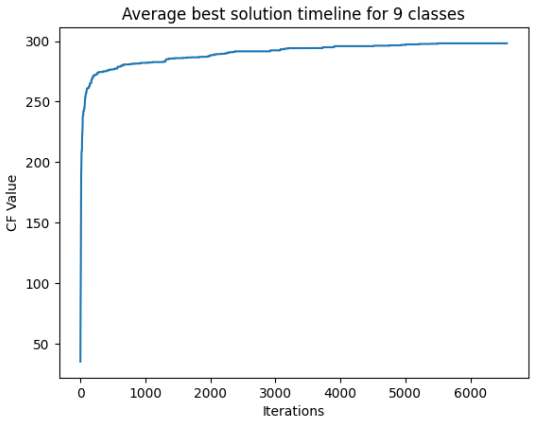
Obrázek 7 – 30 běhů SA, dimenze 9, konfigurace B (900 iterací)



Obrázek 8 – 30 běhů SA, dimenze 9, konfigurace A (6 561 iterací)



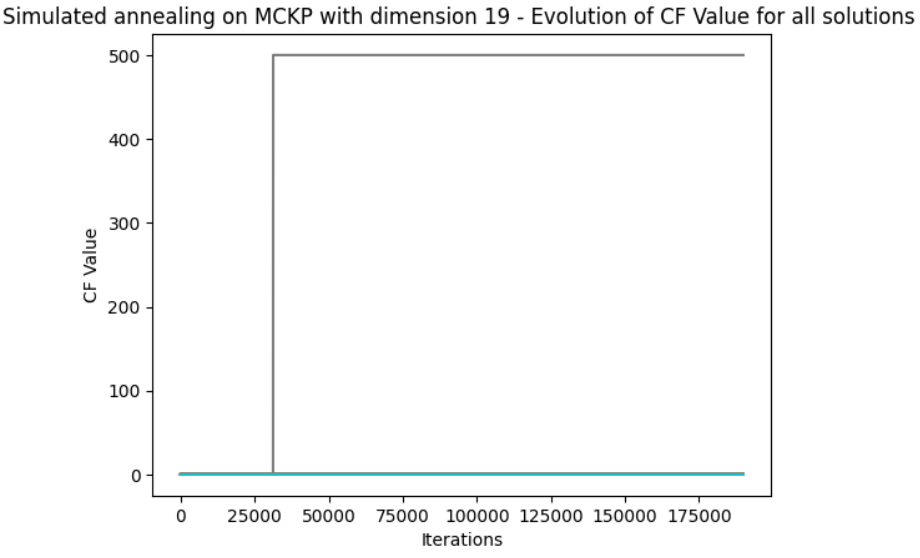
Obrázek 9 – Průměr nejlepších řešení ze 30 běhů SA, dimenze 9, konfigurace B



Obrázek 10 – Průměr nejlepších řešení ze 30 běhů SA, dimenze 9, konfigurace A

Jsou ale také konkrétní množiny předmětů, u kterých je platných možných řešení velmi málo – např. předměty vygenerované pro dimenzi 19, kde je kapacita batohu nastavená na 300, jsou do nejlepšího bruteforcem nalezeného řešení uspořádány tak, že jejich společná hmotnost je 298 a přitom pouze ve 2 z 19 kategorií existují lehčí předměty, které by mohly být použity.

Z více než miliardy všech možných kombinací je v tomto případě platný jen malý zlomek a nalezení platných řešení je tak velmi závislé heuristice. Při posledním spuštění sady 30 běhů SA s konfigurací A (10 000 hlavních iterací v 1 běhu => 300 000 náhodně vygenerovaných počátečních řešení) sice 1 běh řešení našel, ale ve 3 nebo 4 předchozích sadách platné řešení nenalezl žádný. Za těchto okolností si nemyslím, že by prosté zvýšení počtu iterací nebo jiné ladění použité konfigurace dokázalo úspěšnost algoritmu významně vylepšit.



Obrázek 11 – 30 běhů SA, dimenze 19, pouze 1 běh našel platné řešení

4 KÓD POUŽITÝ KE ZPRACOVÁNÍ ÚLOHY

Řešení jsem opět naprogramovala v Pythonu. Funkce a algoritmy jsou implementovány v samostatných souborech (functions.py a algorithms.py). Soubor dataexports.py obsahuje veškeré pomocné metody k vykreslování grafů a exportu dat do souborů.

Při řešení problémů hrubou silou jsem využila rekurzivně volanou metodu k posunutí indexu zkoumaného předmětu o 1:

```
def shift_solution_by_one(original_solution, shift_at_index,
value_limit):
    if shift_at_index < 0:
        return list()
    if original_solution[shift_at_index] + 1 < value_limit:
        original_solution[shift_at_index] += 1
        return original_solution
    original_solution[shift_at_index] = 0
    return shift_solution_by_one(original_solution,
shift_at_index - 1, value_limit)
```

Ukázka kódu 3 – Metoda pro získání následujícího řešení pro bruteforce

Při získávání sousedního řešení pro simulované žíhání pak měním 1 index na o 1 vyšší nebo o 1 nižší (s přetáčením na opačný konec kategorie při přetečení). Který index se mění je určeno náhodně, stejně jako zda se zvětší nebo zmenší:

```
def get_neighbour_solution(centerpoint, dimension,
items_per_category):
    new_solution = list()
    changing_index = random.randint(0, dimension - 1)
    changing_direction = random.randint(0, 1)
    if changing_direction == 1:
        new_value = centerpoint[changing_index] + 1
        if new_value >= items_per_category:
            new_value = 0
    else:
        new_value = centerpoint[changing_index] - 1
        if new_value < 0:
            new_value = items_per_category - 1
    for i in range(len(centerpoint)):
        new_solution.append(centerpoint[i])
    new_solution[changing_index] = new_value
    return new_solution
```

Ukázka kódu 4 – Metoda pro získání sousedního řešení při simulovaném žíhání

Samotná metoda simulovaného žíhání je oproti benchmark úloze upravena z minimalizace na maximalizaci a přizpůsobena práci s vektory cen a vah předmětů.

```
def simulated_annealing(primary_loop_iterations,
metropolis_repetitions,
                        dimension, items_per_category,
item_values_and_weights_tuples, weight_limit,
                        starting_temperature,
min_temperature, cooling_rate):
    very_best_result = float(0)
    very_best_solution = list()
    convergence_list = list()
    best_solution_prices = list()
    best_solution_weights = list()
    temperature = starting_temperature
    start_timestamp = datetime.datetime.now()
    for x in range(primary_loop_iterations):
        centerpoint_solution =
create_filled_random_list(dimension, items_per_category)
        solution_prices_and_weights_tuples =
get_solution_prices_and_weights_tuple_list(centerpoint_soluti
on,

items_per_category,

item_values_and_weights_tuples)
        metropolis_best_result =
knapsack_cost_function(solution_prices_and_weights_tuples,
weight_limit)
        for i in range(1, metropolis_repetitions + 1):
            new_solution =
get_neighbour_solution(centerpoint_solution, dimension,
items_per_category)
            new_prices_and_weights_tuples =
get_solution_prices_and_weights_tuple_list(centerpoint_soluti
on,

items_per_category,

item_values_and_weights_tuples)
            new_result =
knapsack_cost_function(new_prices_and_weights_tuples,
weight_limit)
            if new_result > metropolis_best_result:
                metropolis_best_result = new_result
                centerpoint_solution.clear()
                for nsi in range(dimension):

centerpoint_solution.append(new_solution[nsi])
            else:
                chance_to_keep_previous =
np.random.uniform(0, 1)
                difference = metropolis_best_result -
new_result
```

```
                if chance_to_keep_previous < math.exp(-
difference / temperature):
                    metropolis_best_result = new_result
                    centerpoint_solution.clear()
                    for nsi in range(dimension):

centerpoint_solution.append(new_solution[nsi])
            if metropolis_best_result > very_best_result:
                very_best_result = metropolis_best_result
                very_best_solution.clear()
                for bsi in range(dimension):

very_best_solution.append(new_solution[bsi])
                best_solution_prices.clear()
                best_solution_weights.clear()
                for p in range(dimension):

best_solution_prices.append(new_prices_and_weights_tuples[p][
0])

best_solution_weights.append(new_prices_and_weights_tuples[p]
[1])
            convergence_list.append(very_best_result)
        if temperature * cooling_rate >= min_temperature:
            temperature *= cooling_rate
    end_timestamp = datetime.datetime.now()

de.write_solution_file(f"./output/SA_solution_d{dimension}.cs
v", very_best_solution, best_solution_prices,
                           best_solution_weights,
start_timestamp, end_timestamp)
    return convergence_list
```

Ukázka kódu 3 – Implementace algoritmu Simulated Annealing upravená pro MKCP

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

SEZNAM UKÁZEK KÓDU