# Exaktní a heuristická řešení konstruktivního problému batohu

Marek Nevole
nevolmar@fit.cvut.cz

7. října 2021

**Abstrakt**

Problém 0/1 batohu je jedním z nejznámějších kombinatorických problémů. Cílem tohoto úkolu bylo rozšíření prvního úkolu a implementování pokročilejších metod k řešení. Oproti prvnímu úkolu, kde jsme řešili rozhodovací verzi problému, zde řešíme konstruktivní verzi, ve které hledáme přímo konfigurace, které maximalizují cenovou funkci. Pokročilejšími metodami jsou řešení pomocí dynamického programování, použití hladové heuristiky, rozšíření této heuristiky a poslední metodou bylo využití plně polynomiálně aproximačního schématu, které rozšiřuje metodu dynamického programovaní. Nově implementované metody v drtivé většině případů vykazují lepší výsledky než metody hrubé síly, avšak kvalita výsledku není vždy optimální.

# 1 Úvod

Problém 0/1 batohu je známý kombinatorický problém. Běžná instance tohoto problému je zadána parametry $n$, $M$, kde $n$ je počet dostupných předmětů a $M$ je kapacitu batohu. Dále následuje seznam předmětů, které jsou reprezentovány dvojicemi. Dvojice se skládá ze dvou čísel. První udává váhu předmětu a druhé jeho hodnotu/cenu. Pro účely tohoto papíru bereme v potaz konstruktivní verzi problému, ve které je úkolem poskládat věci do batohu tak, aby součet vah věcí nepřesahoval kapacitu batohu a zároveň součet hodnot věcí byl maximalizován. Odkaz na celé zadání zde.

# 2 Implementované metody

V této části jsou popsané implementované metody, a k nim přiložené pseudokódy.

## 2.1 Metoda větví a hranic

Metoda větví a hranic je jednoduché rozšírení algoritmu hrubé síly, které výrazně prořeže a tím zmenší stavový prostor, který musí být následně prohledán. Implementace této metody pro konstruktivní verzi problému, oproti rozhodovací, je rozšířena o ukládání nejlepšího současně dosaženého řešení. Při rekurzivním průchodu stavového prostoru je poté kontrolováno, zda pomocí

zbývajících předmětů lze překonat cenu současného nejlepšího řešení. Pokud je hodnota zbylých věcí při součtu s dosavadní cenou batohu menší než cena nejlepší dosud nalezeného řešení, tak nemá smysl prohledávat tuto větev stavového prostoru a algoritmus se vrací.

**Algorithm 1:** BacktrackBNB(item_id, price, weight, remaining_price)

```
1  if weight > max_capacity then
2  |  return False;
3  end
4  if best_price < price then
5  |  best_price ← price;
6  |  best_solution ← solution;
7  |  best_weight ← weight;
8  end
9  if price + remaining_price <
     best_price then
10 |  return False;
11 end
12 if ran out of items then
13 |  return False;
14 end
15 # Add current item;
16 solution[item_id] = 1;
17 BacktrackBNB(item_id + 1, price +
     item_price, weight + item_weight,
     remaining_price - item_price);
18 # Don't add current item;
19 solution[item_id] = 0;
     BacktrackBNB(item_id + 1, price,
     weight, remaining_price -
     item_price);
```

## 2.2 Hladová heuristika

Hladová heuristika je velice jednoduchý způsob, kterým lze řešit problém batohu. U algoritmu ovšem není zaručeno, že vždy vratí optimální maximalizující řešení. Principem heuristiky je seřadit předměty podle poměru ceny/váhy od největšího a postupně je přidávat do batohu, pokud není přesažena kapacita batohu. Po lineárním průchodu přes všechny věci je vráceno řešení.

**Algorithm 2:** Greedy(sorted_items)

```
1  weight = 0;
2  price = 0;
3  solution = bit vector filled with
     zeroes;
4  foreach item in sorted_items do
5  |  if item.weight + weight <=
   |    capacity then
6  |  |  weight + = item.weight;
7  |  |  price + = item.price;
8  |  |  solution[item.original_index]
   |  |    = 1;
9  |  end
10 end
11 return solution, price, weight
```

## 2.3 Redux rozšíření

Hladová heuristika má hned několik slabin. Jednu z nich řeší rozšíření této heuristiky zvané Redux. Redux je nadstavbou hladové heuristiky. Příklad, na kterém hladová heuristika selhává, lze najít zde. Princip fungování je opět jednoduchý. Redux nejprve zavolá hladovou heuristiku, která

vrátí výsledek. Tento výsledek je poté porovnáván s jedním nejdražším předmětem, který se do batohu vejde. Vráceno je lepší ze dvou řešení.

**Algorithm 3:** Redux(sorted_items)

```
1 grdy_sol = Greedy(sorted_items);
2 rdx_sol =
    priciest_item_that_fits_in_bag;
3 return Better solution in terms of
    price of the two
```

## 2.4 Dynamické programování

Zcela jiným přístupem je dynamické programování. Dynamické programovaní využívá paměti k memoizování již provedených výpočtů a tedy není nutné provádět stejné výpočty vícekrát, což na úkor paměti, při správném použití, ušetří mnoho výpočetního času. Dynamické programovaní je vždy spojeno s rozkladem neboli dekompozicí problému na menší problémy. Pro problém batohu dle dekompozice podle ceny je vytvořeno pole o velikosti počet předmětů + 1 × součet cen předmětů + 1. Tabulku $W$ vyplníme podle následujících pravidel:

$W(0,0) = 0$,
$W(0,c) = \infty$ pro všechna $c > 0$,
$W(i+1,c) = min(W(i,c), W(i,c-c_{i+1}) + w_{i+1})$ pro všechna $i > 0$, kde $w_i$ je váha i-tého předmětu a $c_i$ je cena i-tého předmětu.

Každou buňku tabulky navštívíme právě jednou, tedy složitost tohoto algoritmu se odvíjí podle počtu předmětů, ale hlavně podle součtu cen všech předmětů, který je vždy vyšší než počet předmětů. Složitost činí O($n$∗součet cen).

**Algorithm 4:** DP(items, sum_of_prices)

```
1 initialize W;
2 W[0][0] = 0;
3 W[0][c] = ∞ for c ¿ 0;
4 W[i+1][c] = min(W[i][c], W[i][c -
    c_{i+1}]+w_{i+1}) for c>=0 and i > 0;
5 Find best price and weight in last
    column of W table;
6 Construct solution vector from W
    table;
7 return price, weight, solution vector
```

## 2.5 FPTAS algoritmus

Problém batohu lze řešit pomocí plně polynomiálního časově aproximačního schématu (FPTAS). Tedy tento problém patří mezi nejjednodušší NP-úplné problémy. Aby byl algoritmus FPTAS, musí být použita dekompozice podle ceny z dynamického programování. Použitý algoritmus využívá předchozí algoritmus, akorát před samotným spuštěním upravuje ceny všech předmetů podle následujících vzorců:

$K = \frac{\epsilon * c_{max}}{n}$
$c_i = \lfloor \frac{c_i}{K} \rfloor$

Při vhodně zvoleném parametru $\epsilon$, tak aby $K > 1$, se zmenší počet "cenových kategorií"neboli součet cen předmětů, a tedy tabulka dynamického programování nemusí být tak velká a výpočet je značně rychlejší. Zároveň lze poté použít tento algo-

ritmus i na instance, které zadávají hodnotu předmětů z reálných čísel, protože překalkulované ceny předmětů jsou zaokrouhleny dolů. Ovšem se zvětšujícím parametrem $\epsilon$ roste nepřesnost algoritmu a relativní chyba oproti optimálnímu řešení, neboť je možné, že předměty s odlišnou cenou skončí ve stejné "cenové kategorii". Naopak pokud se $K$ přibližuje k 1 zleva, tak ztrácíme časovou výhodu oproti algoritmu, který využívá běžné dynamické programování. Výpočetní složitost činí $O(n^2\lfloor\frac{n}{\epsilon}\rfloor)$.

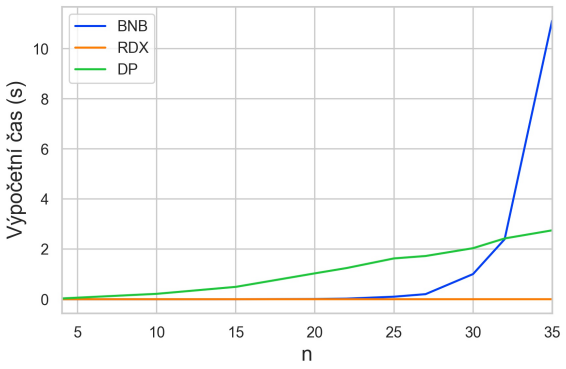
**Algorithm 5:** FPTAS(items, $\epsilon$)

1 $K = \frac{\epsilon * c_{max}}{n}$;
2 new_items = Recalculate price of each item according to $c_i = \lfloor\frac{c_i}{K}\rfloor$;
3 solution = DP(new_items);

# 3 Experimenty

Pro testování byly vygenerovány tři testovací sady N, ZKC a ZKW. Sada N byla zcela náhodně vygenerována. O generování sad ZKC a ZKW nám nebylo nic sděleno, ale v průběhu experimentů je zřejmé, že byly generovány, tak aby objevily slabiny implementovaných metod. Všechny sady obsahují stejný počet testů, co se parametru $n$ týče, konkrétně $n = 4$ 10 15 20 22 25 27 30 32 35 37 40. Čas řešení byl měřen jako system + CPU time, z výsledného času je vynechán sleep time. Testy byly spuštěny na následující platformě: Intel Core i5-3350P 4 Cores 3.10 GHz, Windows 10, Python 3.8.8.

Z obrázku 1 lze pozorovat průměrné výpočetní časy uvedených metod v závislosti na velikostí sad instancí z




Obrázek 1: Graf průměrného výpočetního času algoritmů branch&bound, heuristiky Redux a dynamického programování v závislosti na velikosti instancí dle $n$ z náhodně generované sady dat.

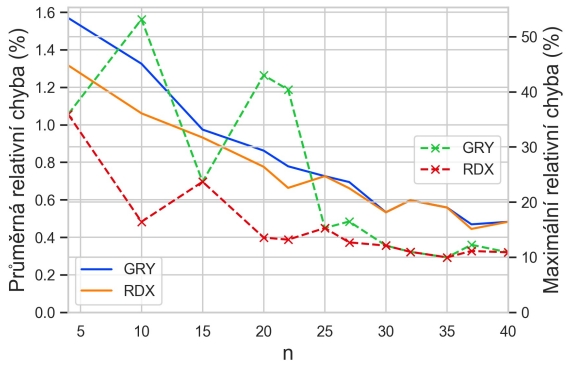
náhodně generované sady dat. Algoritmus B&B a algoritmus využívající dynamického programování vždy vráti optimální řešení. Metoda B&B má exponenciální složitost v závislosti na parametru $n$, což lze z tohoto obrázku pozorovat, kde přibližně od $n = 30$ výpočetní čas řešení přesahuje 1 vteřinu a exponenciálně roste dál. Jak bylo zmíněno v implementačních detailech dynamického programovaní, tak časová složitost roste s n a součtem ceny předmětů. Podrobnější analýza této složitosti je k obrázku 4. Složitost hladové heuristiky a heuristiky redux záleží na řadícím algoritmu, který je v pythonu implementován jako timsort, jehož složitost je v nejhorším případě $O(nlogn)$. Samotný průchod přes všechny předměty je poté lineární, a jelikož počet předmětů je rovný $n$, tedy maximálně 40, tudíž výpočetní čas těchto metod bude vždy až několikasetkrát nižší něž u ostatních. Ovšem tyto metody nezaručují optimální řešení a tímto se zabývá další

experiment.

V pořadí 2. experiment se zabývá kvalitou vráceného řešení hladové heuristiky a jejího rozšíření Redux. Na náhodně generovaných datech ze sady N byl proveden test, jehož výsledky jsou pozorovatelné na obrázku 2. V tomto testu byla změřena průměrná a maximální relativní chyba v závislosti na velikosti instancí. Relativní chyba byla vypočítána jako rozdíl optimálního a vráceného řešení heuristikou vydělený optimálním řešením a vynasobený 100, aby výsledek byl v procentech.

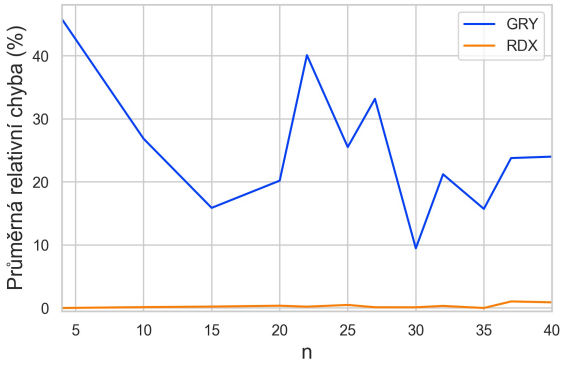
$$rel\_err = \frac{(opt - heur\_sol)100}{opt}$$

Z obrázku 2 lze pozorovat, že maximální i průměrná relativní chyba, při testování větších instancí, klesá. Průměrná relativní chyba se pohybuje od přibližně 0.4 % do 1.6 %. Maximální relativní chyba je až na pár vyjímek pro oba algoritmy téměř shodná. Nejdůležitějším závěrem z tohoto testu je, že ve všech případech je Redux lepší a druhý lineární průchod přes seřazené předměty ho zpomaluje zanedbatelně oproti řazení.




Obrázek 2: Graf průměrné a maximální relativní chyby v %, v závislosti na velikosti instancí na náhodně generovaných datech.

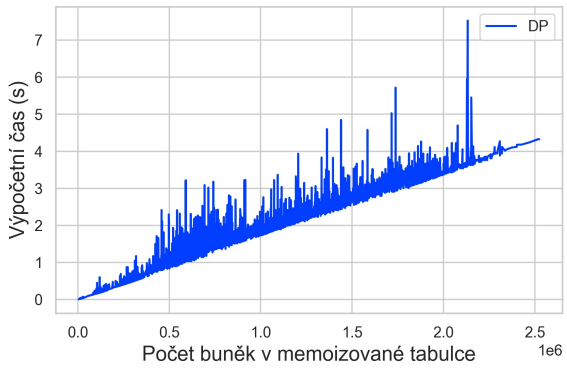
Dalším testem bylo porovnání relativních chyb heuristik na testovací sadě ZKW. Výsledky tohoto testu jsou na obrázku 3. Zde se výsledky, oproti předchozímu testu, výrazně liší. Po krátkém nahlédnutí do testovací sady je zřejmé, že značná část instancí je vygenerována, tak aby zvýhodňovala heuristiku Redux. Tedy, že optimální řešení těchto instancí se skládá z jednoho jediného předmetu, jehož poměr cena/váha není největší a první, který se do batohu vejde. Průměrná relativní chyba heuristiky Redux je téměř nulová, zatímco tato chyba u hladové heuristiky se pohybuje mezi 10 až 50 %.



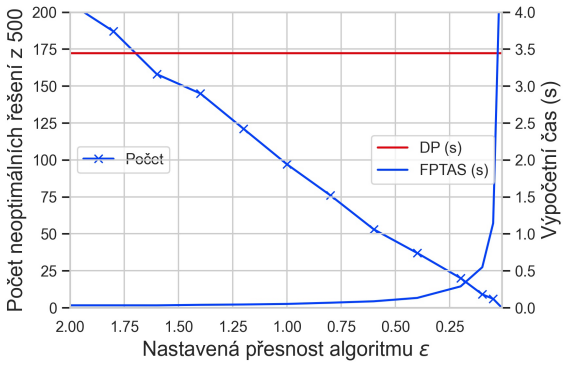

Obrázek 3: Graf průměrné a maximální relativní chyby v %, v závislosti na velikosti instancí na testovací sadě ZKW.

Obrázek 4 se zaobírá analýzou výpočetní složitosti metody dynamického programovaní. Jak bylo zmíněno složitost této metody úzce souvisí s počtem hodnot v memoizační tabulce. Na tomto obrázku lze sledovat výpočetní čas závislý na počtu buněk v tabulce. Test byl proveden na všech instancích v náhodně generovaných datech.

Poslední provedený experiment se týka FPTAS algoritmu. Testy byly provedeny




Obrázek 4: Graf výpočetního času (s) v závisloti na počtu buněk v memoizační tabulce pro náhodně vygenerovaná data.




Obrázek 5: Graf počtu neoptimálních řešení a výpočetního času (s) v závislosti na zvolené přesnosti (parametr $\epsilon$) na náhodně generovaných instancích o velikosti instance $n = 40$. V porovnání s průměrným výpočetním časem těchto instancí metodou dynamického programování.

pouze na náhodně vygenerovaných instancích o velikosti $n = 40$. Na obrázku 5 lze pozorovat, že výpočetní čas FPTAS algoritmu je několika násobně nižší na úkor optimality výsledku. Vhodné zvolení $\epsilon$ záleží na tom, co od algoritmu chceme. Pokud chceme aproximovat řešení a relativní chyba je do určité míry v pořádku, tak volíme větší hodnoty. Pokud se chceme přiblížit k optimalitě, a zároveň chceme rychlejší vypočetní čas oproti pouhému dynamickému programování, volíme $\epsilon$, tak aby se hodnota $K$ blížila k 1 zprava. Na obrázku 5 je možné si všimnout, že při zvolení $\epsilon = 0.01$ bylo výsledné $K$ v několika případech $< 1$, a rekalkulace cen ceny zvedla a průměrný výpočetní čas byl větší než u metody dynamického programovaní, ovšem chybovost byla samožřejmě nulová.

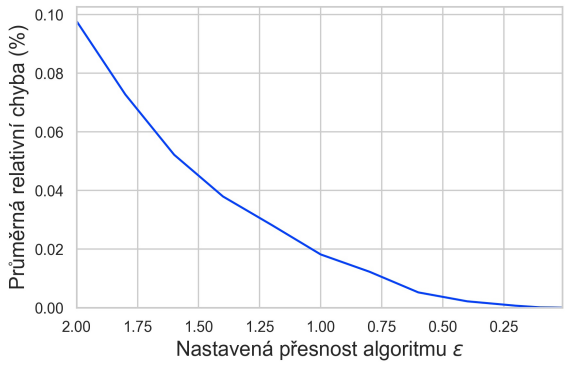
Obrázek 6 je pouze doplněním testu z obrázku 5. Lze pozorovat, že průměrná relativní chyba i při $\epsilon = 2.0$ je pouze 0.1%. Průměrný výpočetní čas je přibližně 0.03 s, tedy více jak 100 násobně rychlejší než u DP.

# 4 Závěr

V tomto úkole bylo implementováno 5 metod, jak řešit konstruktivní verzi kombinatorického problému 0/1 batohu. Těmito metodami byly metoda větví a hranic, hladová heuristika, heuristika Redux, metoda využívající dynamické programování a algoritmus klasifikovaný jako plně polynomiálně aproximační schéma, založené na metodě dynamického programovaní.

Z provedených experimentů je patrné, že metoda větví a hranic z prvního úkolu na tyto metody nestačí. I přes její optimalitu je výpočetní čas stále exponenciální a na velkých instancích je značně pomalejší.

Hladová heuristika a heuristika Redux jsou výrazně rychlejší až několikasetkrát. Ovšem nevracejí optimální řešení, které je občas vyžadováno. Dle testů vyobrazených na obrázku 3, lze generovat instance,




Obrázek 6: Graf průměrné relativní chyby (%) v závislosti na zvolené přesnosti (parametr $\epsilon$) na náhodně generovaných instancích o velikosti instance $n = 40$.

které dramaticky zvednou průměrnou a maximální relativní chybu řešení těchto heuristik.

Metoda dynamického programování zaručuje optimalitu, ale je silně zpomalena na instancích, jejichž součet cen předmětů je řádově vyšší než celkový počet předmetů, díky velikosti memoizační tabulky.

Na nedostatky předchozí metody reaguje FPTAS algoritmus, který nejen zrychluje výpočet (při správně zvoleném $\epsilon$), ale také umožňuje řešit instance, jejichž předměty mají cenu z oboru reálných čísel. Z celého názvu FPTAS je patrné, že se jedná o aproximační algoritmus, který nemusí vracet optimální řešení, avšak z testů je pozorovatelné, že při 100 násobném zrychlení byla relativní chyba pouze 0.1%.