Domácí úloha č. 3 – Knapsack problém II.

Specifikace úlohy

Cílem domácí úlohy bylo pokračování v první části úlohy rozšířením původní metody hrubou silou o další metody řešení. Konkrétně řešení metodou větví a hranic, metodou dynamického programování a aproximativním algoritmem.

Nástroje k řešení

K implementaci jsem využil programovací jazyk **Java** pod prostředím **NetBeans**. Všechny výpočty běželi na procesoru Intel Core 2 Duo 3.00 GHz a pod operačním systémem Microsoft Windows 7. Výsledky byly zpracovány tabulkovým procesorem Microsoft Excel.

Výsledný zdrojový kód je spouštěn ze souboru **Main.java**, zbytek kódu je přehledně rozdělen do tříd.

K měření času jsem použil funkci **System.currentTimeMillis().**

Rozbor variant řešení

Úkolem bylo:

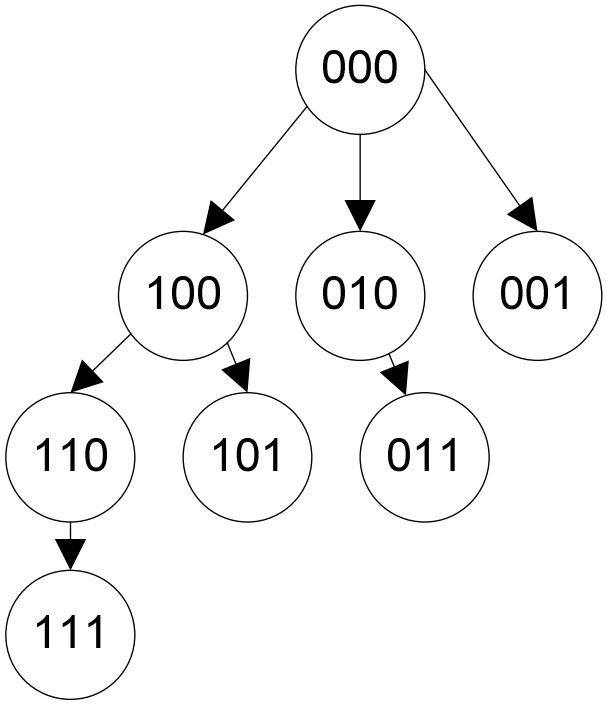
- Naprogramovat řešení problému **metodou větví a hranic, tak aby omezujícím faktorem byla hodnota optimalizačního kritéria** a porovnat výpočetní čas s ostatními metodami.
- Naprogramovat řešení problému **metodou dynamického programování** s dekompozicí podle váhy, nebo ceny.
- Modifikovat řešení s metodou dynamického programování tak, aby pracoval s omezenou přesností, tzv. **aproximativní algoritmus**.

Popis postupu řešení

Algoritmus hrubou silou

Tento algoritmus jsem použil z první úlohy. Vzhledem k tomu, že jsem ho prvotně implementoval pomocí iterace přes všechny možné stavy, musel být celý přepracován. Nový algoritmus hrubou silou tedy již pracuje se zásobníkem a procházení je řešeno rekurzivně. Lze tak daleko lépe ořezávat stavoví prostor, v tomto případě expandující větev odřízneme, pokud je její aktuální cena větší, než zatím nejlepší řešení.

Expanze probíhá podle dalšího diagramu a je důležitý i pro následující algoritmus, proto ho zmiňuji:



Expanze probíhá tak, že vždycky najdu jedničku, která je nejvíce vpravo a zbytek nul doplním střídavě jedničkami. U prvního stavu tedy rovnou doplňuji jedničky postupně na každou pozici a nalezené stavy dále expanduji. Tímto je zaručeno, že se na zásobník mohou dostat všechny možné stavy, ale zároveň lze celou větev odříznout, protože potomek bude vždy těžší (bude mít vždy vyšší cenu).

Složitost algoritmu je dána počtem stavů a je 2^N (protože každá věc v batohu buď je, nebo není).

Algoritmus metodou větví a hranic

Tím, že jsem si algoritmus hrubé síly přepsal do rekurzivního zpracování, mohl jsem ho použít přímo jako základ pro algoritmus větví a hranic. U algoritmu hrubé síly jsem stavový prostor ořezával pouze ořezávání shora (nelze překročit kapacitu batohu) a u algoritmu větví a hranic doplním ještě ořezávání se spodu (stávající řešení nemůže být už lepší, než aktuální nejlepší nalezené).

To, jestli může být řešení ještě lepší, zjistím pomocí funkce *isStateSuitable()*, kterou sem doplnil do rekurzivního zpracování. Ta opět využívá diagramu, který je nakreslen výše. V každém

stavu opět najdeme jedničku, která je nejvíce vpravo. Zbytek stavů doplníme jedničkou (jako by zbytek věci v batohu byl). Tyto doplněné stavy jsou ty, o kterých jsem ještě nerozhodnul, zdali v batohu budou, nebo ne. Tímto postupem získáme maximální možnou dosažitelnou cenu, které se v dané větvi stromu může dosáhnout.

Algoritmus je opět úplný a složitost zůstává O(2^n), ale díky ořezávání dosahuje lepších časů, jak uvidíme na naměřených datech níže.

Algoritmus dynamického programování

Tato metoda výpočtu je opět rekurzivní, ale v tomto případě jsem jí implementoval tak, že funkce volá sama sebe. Dekompozici jsem prováděl **podle kapacity batohu**, tak jak jsme to dělali na cvičení. Celkově je můj algoritmus přepisem tohoto pseudoalgoritmu:

```
function KNAP (V, C, M)
    if isTrivial(V, C, M) return trivialKNAP(V, C, M);
    (X0, C0, m0) = KNAP(V-{vn}, C-{cn}, M);
    X1, C1, m1) = KNAP(V-{vn}, C-{cn}, M-vn);
    if (C1+cn) > C0 return(X1.1,  C1+cn, m1+vn);
       else return(X0.0, C0, m0);

function isTrivial(V, C, M)
    return(isEmpty(V) or M=0 or M<0);
```

Algoritmus má dvě fáze. První dopředná fáze začíná s plným batohem a postupně nám rozbaluje rekurzivní strom tak, že volá jednu větev, kde daná věc je (`KNAP(V-{vn}, C-{cn}, M-vn)`) a druhou větev, kde daná věc není (`KNAP(V-{vn}, C-{cn}, M)`). Ve zpětné fázi se určuje, které větev je lepší (dle dosažené ceny) a podle toho se vrátí výsledek větve, kde věc byla, nebo ne. Zároveň je důležité kontrolovat, aby se nepřesáhla kapacita batohu.

Další věcí, které se u dynamického programování řeší, je ukládání již vypočítaných stavů. Funguje to tak, že každý stav definuji dle aktuálního počtu položek a nosnosti batohu, která je ještě k dispozici. Ukládané stavy jsou vlastně jedna konfigurace, neboli obsah batohu a jeho zbývající nosnost. Výhodou tohoto ukládání je, že pokud se stav již jednou počítal, není nutné ho počítat opětovně.

Složitost algoritmu je dána rozměrem pomyslné „tabulky“, které definuje rekurzivní sestup. Na ose x jsou jednotlivé věci batohu a na ose y je zbývající kapacita. Složitost algoritmu je tedy O(M.n), ale má vysoké paměťové nároky, protože je nutné uchovávat již vypočítané stavy.

Aproximativní algoritmus

Tento algoritmus vychází z algoritmu dynamického programování, ale provádí aproximaci tím, že snižuje možný počet celkových stavů. Pokud zvolíme například aproximační konstantu 2, algoritmus provede snížení kapacity batohu na polovinu a stejně tak u všech věcí se provede snížení hmotnosti na polovinu. Tím se dosáhne určitého zaokrouhlení a snížení počtu možných stavů na polovinu. Volbou aproximační konstanty lze měnit dobu potřebnou k výpočtu a dále relativní chybu výpočtu. Čím větší konstanta, tím méně procházených stavů, tím lepší čas a větší chyba. Relativní chybu vypočítám stejně jako u první úlohy a to v závislosti k referenčnímu (správnému) řešení.

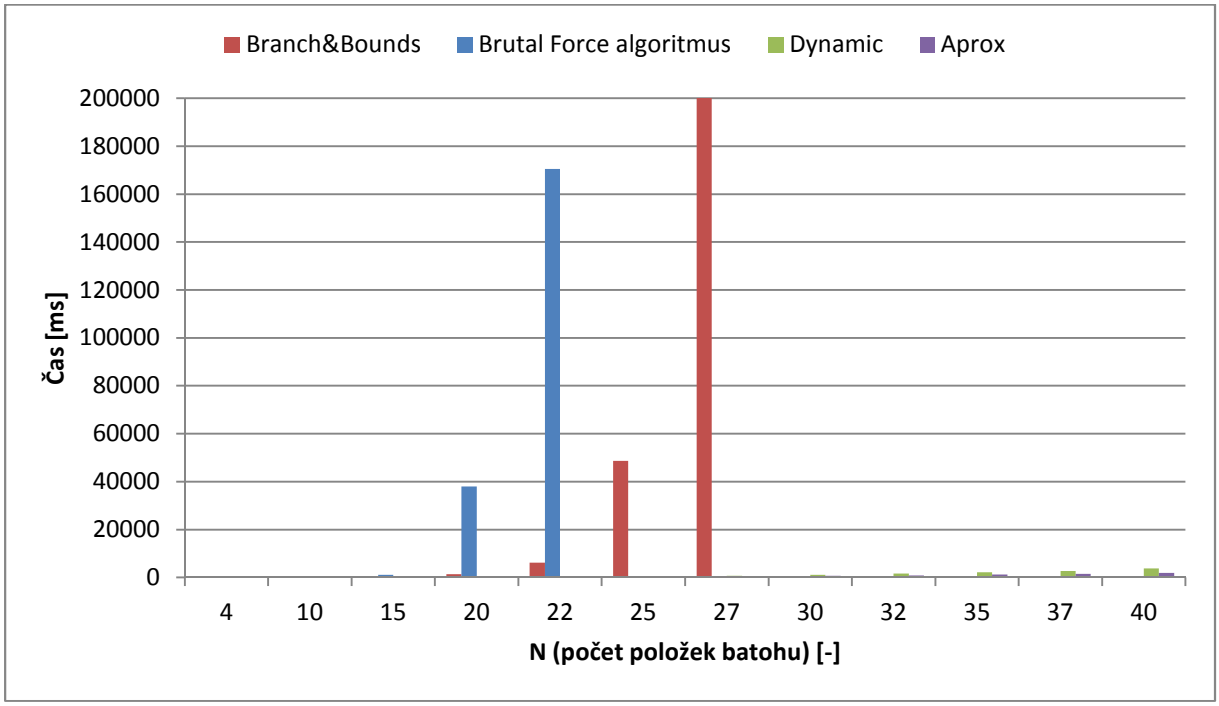
Výpočetní časy a grafy

Výpočetní časy jednotlivých řešení

V následující tabulce uvádím výpočetní časy jednotlivých řešení. Pro malé instance byl algoritmus proveden několikrát v cyklu, aby došlo k přesnějšímu výsledku. Časy jsou pak přepočteny na jeden průchod (v pravé části tabulky). Aproximativní algoritmus byl měřen pro konstantu 2.

N	# opak	BruteForce	B&B	Dynamic	Aprox (k=2)	BruteForceOne	B&B	Dynamic	Aprox (k=2)
4	1000	5303	6492	717	680	5,303	6,492	0,717	0,68
10	1000	30207	11980	13563	10350	30,207	11,98	13,563	10,35
15	500	522451	20971	41656	26204	1044,902	41,942	83,312	52,408
20	10	380046	13835	2531	1501	38004,6	1383,5	253,1	150,1
22	1	170461	6199	471	200	170461	6199	471	200
25	1		48639	607	369	0	48639	607	369
27	1		204087	765	484	0	204087	765	484
30	1			1117	706			1117	706
32	1			1579	875			1579	875
35	1			2181	1189			2181	1189
37	1			2705	1460			2705	1460
40	1			3713	1942			3713	1942

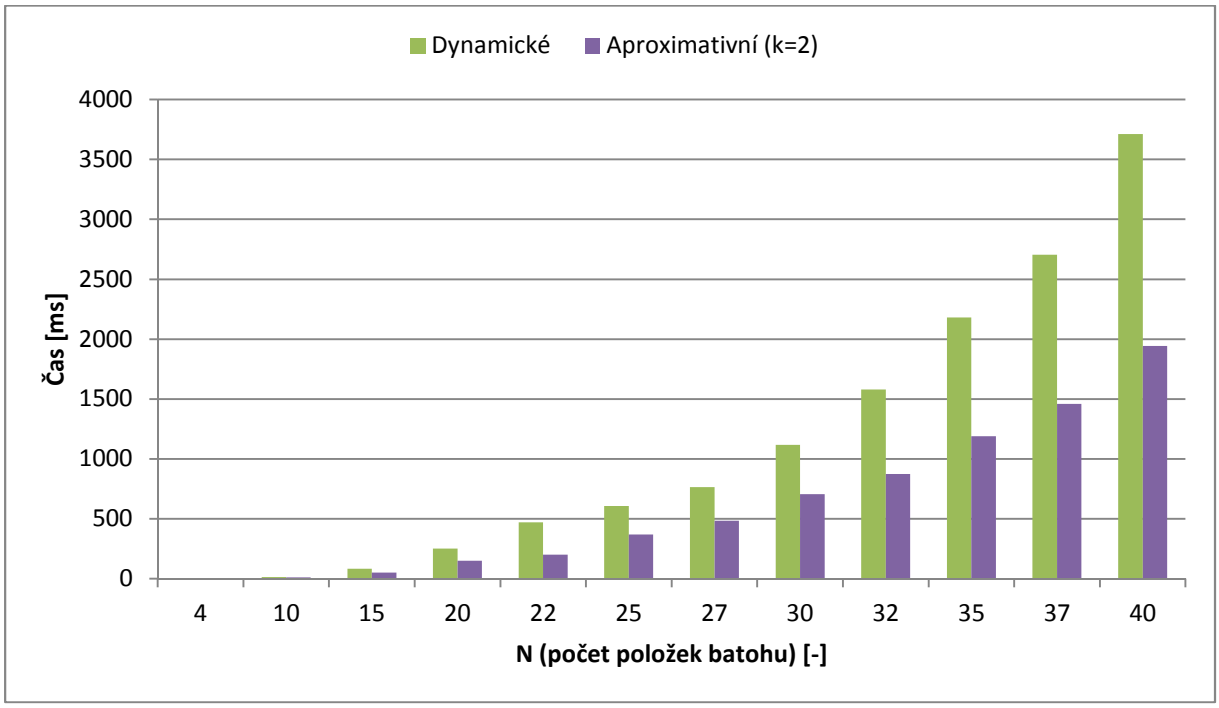
Grafické znázornění tabulky:



Na grafu je vidět, že časově nejnáročnější je algoritmus hrubou silou a poté B&B, který má nárůst ale opožděný z důvodu lepšího ořezávání stavového prostoru.

Výpočetní časy dynamického a aproximativního

Znázornění výpočetních časů bez BrutalForce algoritmu a bez B&B, aby byla vidět závislost mezi dynamickým algoritmem a aproximativním pro konstantu 2 (na horním grafu to není zřejmé).



Závislost chyby a výpočetního času aproximativního algoritmu

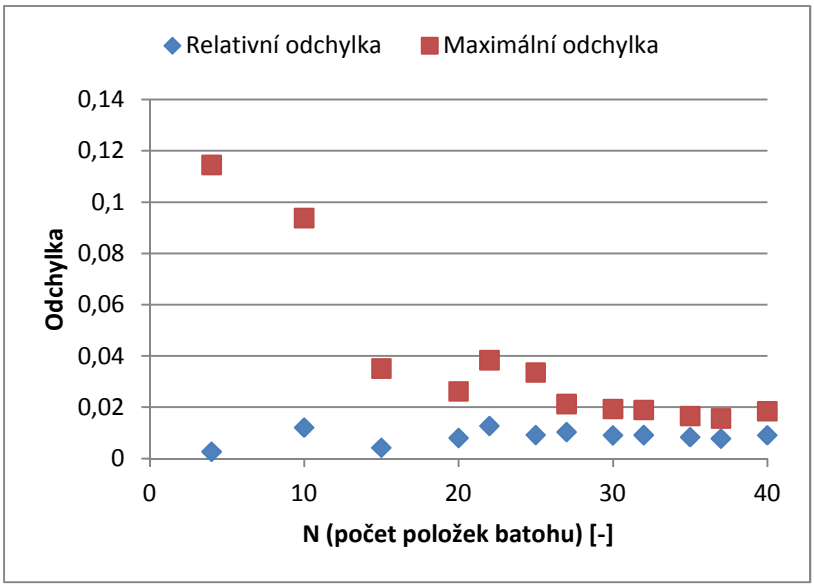
S rostoucí aproximační konstantou nám klesá počet možných stavů, klesá výpočetní čas, ale roste chybovost.

Relativní odchylka v závislosti na instanci

Pro zajímavost jsem si zkusil změřit, jak se mění relativní odchylka v závislosti na instanci. Měření probíhalo pro aproximační konstantu 2.

N	Relativní odchylka	Maximální odchylka
4	0,0028	0,11463415
10	0,0122	0,09390126
15	0,0043	0,0352229
20	0,0081	0,02630166
22	0,0128	0,03844492
25	0,0093	0,03365236
27	0,0104	0,02137547
30	0,0091	0,0194012
32	0,0092	0,01902399
35	0,0084	0,01666667
37	0,0079	0,01571165
40	0,0092	0,01853976

Grafické znázornění:



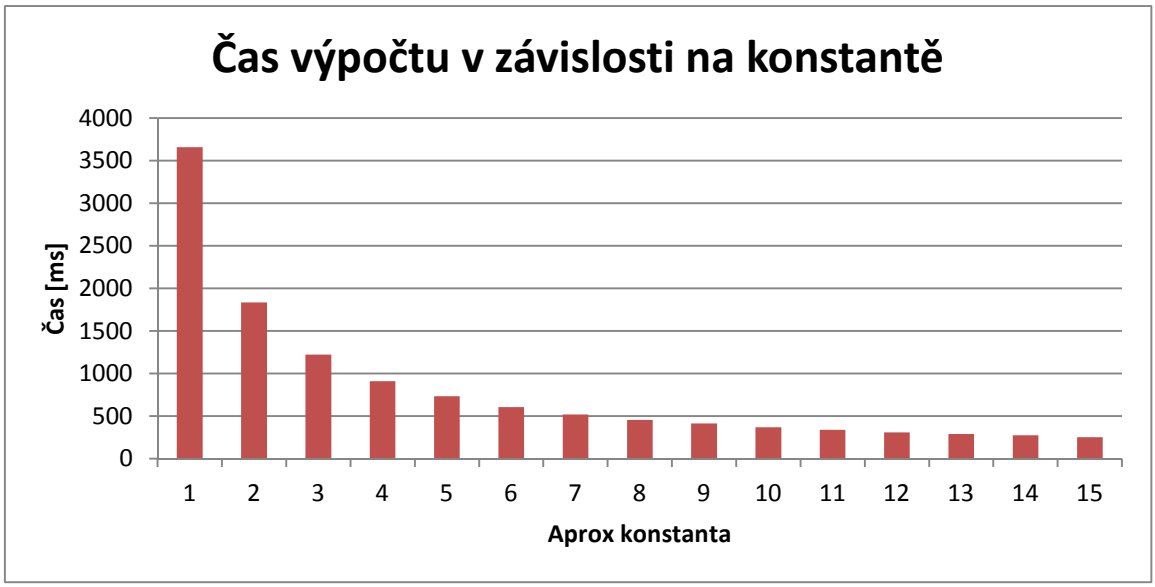
Na grafu je vidět, že relativní odchylka je přibližně pořád stejná, pouze u instancí s nízkým počtem věcí je vyšší maximální odchylka, protože čím méně máme věcí, tím větší chybu způsobí přítomnost/nepřítomnost jedné věci.

Relativní a maximální odchylka v závislosti na zvolené konstantě

Nyní budeme pozorovat, jak se změní výpočetní čas a relativní odchylka v závislosti na změně aproximační konstanty. Měření probíhalo již pouze pro instanci č. 40, tzn. největší možná.

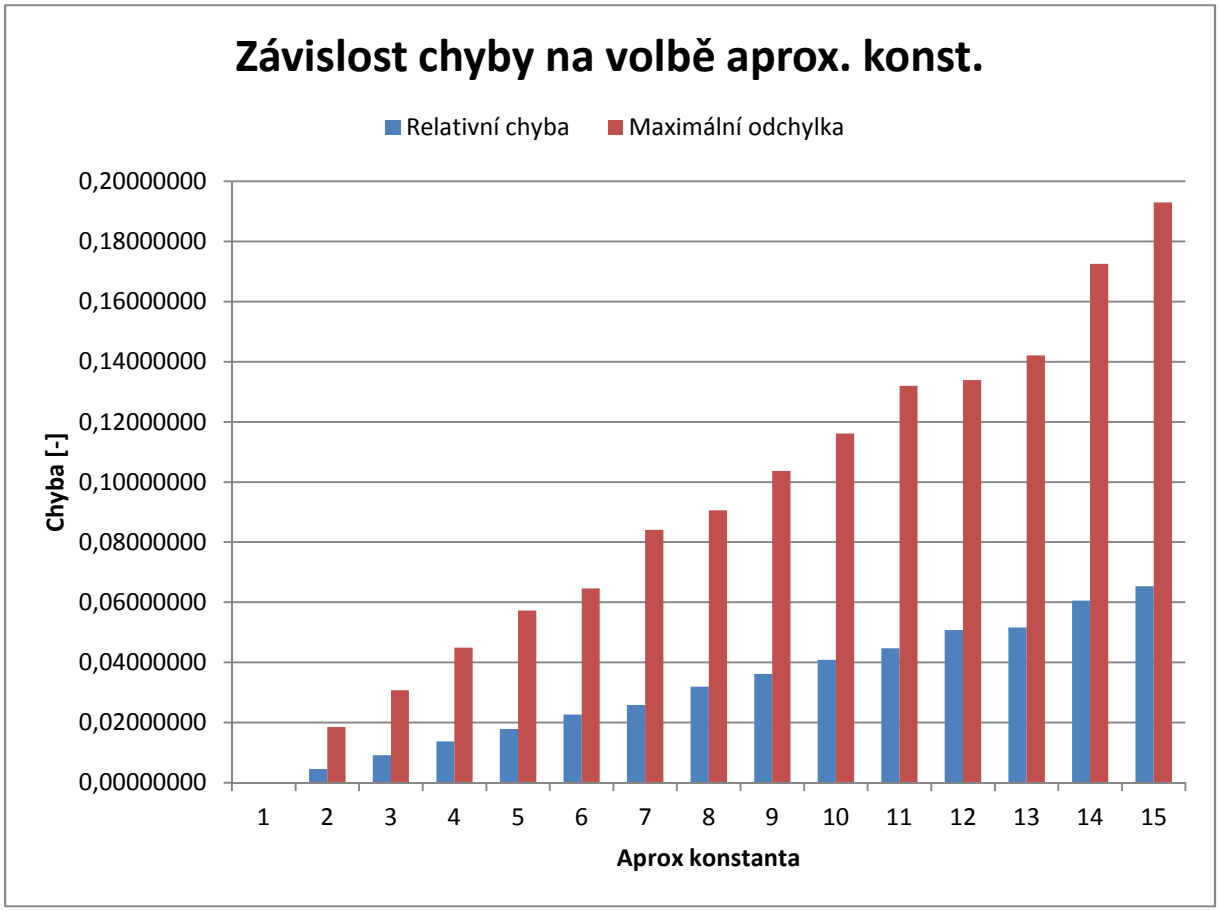
Konstanta	čas výpočtu [ms]	relativní chyba	maximální chyba
1	3656	0.0	0.0
2	1832	0,00459165	0,01853976
3	1221	0,00912701	0,03076292
4	908	0,01366437	0,04484979
5	730	0,01785403	0,05729614
6	605	0,02262854	0,06460245
7	516	0,02580240	0,08412017
8	454	0,03190807	0,09059633
9	411	0,03618294	0,10364807
10	365	0,04085751	0,11609442
11	336	0,04468989	0,13197425
12	306	0,05077994	0,13390558
13	289	0,05162425	0,14214613
14	271	0,06057727	0,17259174
15	251	0,06534624	0,19304281

Grafické znázornění závislosti výpočetního času a konstanty:



Je krásně vidět, že pokud zvyšujeme konstantu, tím zvyšujeme zaokrouhlení a snižujeme možný počet stavů. Klesá tím čas výpočtu.

Grafické znázornění závislosti volby aproximační konstanty na relativní a maximální chybu.



Je vidět, že relativní i maximální chyba roste se zvolenou aproximační konstantou.

Závěr

Implementací dalších metod pro řešení batohu jsme si ověřili, že ořezávání zdola i shora nám sice prohledávaný prostor zmenší, ale ne natolik. Daleko lepších výsledků lze dosáhnout pomocí dynamického programování, nebo jeho aproximační varianty, která už ovšem zrychluje na úkor přesnosti.