Řešení problému batohu dynamickým programováním, metodou větví a hranic a aproximativním algoritmem

1 SPECIFIKACE ÚLOHY

Cílem tohoto úkolu bylo naprogramovat řešení problému batohu a to metodou dynamického programování, metodou větví a hranic a aproximativním algoritmem. Zpráva také obsahuje srovnání těchto algoritmů z hlediska časového výkonu a v případě aproximativního algoritmu pak diskuzi závislosti chyby na výpočetním času a srovnání teoretické chyby s reálně naměřenou. Více informací viz odpovídající stránka na Eduxu.

2 ROZBOR MOŽNÝCH VARIANT ŘEŠENÍ

V části řešení pomocí **dynamického programování** jsme se mohli rozhodnout, zda problém řešit odshora s dopřednou a zpětnou fází, anebo odzdola pouze se zpětnou fází.

V prvním případě bychom postupovali od cílové úlohy, ale protože potřebné podvýsledky nejsou ještě k dispozici, postupně bychom se zanořovali až k triviálním podúlohám, které bychom vyřešili. Odtud bychom zahájili zpětnou fázi a z již známých podřešení konstruovali řešení nadúloh.

Já jsem zvolil řešení odzdola a mám tedy pouze zpětnou fázi. V tomto řešení začínáme řešit problém od triviálních problémů a postupně postupujeme směrem nahoru k řešení složitějších úloh za použití výsledků podúloh.

Také bylo možné se rozhodnout, zda provést dekompozici podle celkové ceny nebo kapacity batohu. Zvolil jsem celkovou cenu, abych mohl řešení použít u třetí části úlohy.

3 POPIS POSTUPU ŘEŠENÍ A ALGORITMU

Program nejdříve načte data ze vstupních souborů a namapuje data z textové podoby na interní objekty, jako jsou např. **KnapsackInstance** či **KnapsackItem**.

3.1 METODA VĚTVÍ A HRANIC (B&B)

0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

Klíčem řešení metodou B&B je generátor kombinací, který např. pro instanci problému o velikosti třech věcí vygeneruje kombinace viz tabulka vlevo. Je to z toho důvodu, protože tato metoda je založena na optimalizaci metody hrubé síly.

O vyřešení konkrétního problému se stará třída **BaBKnapsackSolver**, které je předána instance problému. Tento solver pracuje téměř stejně, jako **BruteKnapsackSolver** (viz předchozí zpráva). Základní rozdíl zde je však v tom, že při detekci kombinace, která batoh přetěžuje (nebo nemůže mít lepší cenu), dojde k „odříznutí“ dané větve kombinací, místo toho, aby byla pouze přeskočena daná kombinace.

O „odřezávání“ se stará metoda **cutBranch**, která na vstupu dostává kombinaci, kterou je třeba odříznout. Příklad takové kombinace: **0-1-1-1-0-1-0-0-0-0**. Metoda nalezne poslední výskyt jedničky, čímž identifikuje kořen předmětů. V tuto chvíli již víme, že přidávání jakýchkoliv dalších předmětů nemůže batohu nijak pomoci (jak to víme, viz níže), a proto nastaví všechny tyto zbývající pozice na hodnotu 1, čímž v našem příkladu vznikne: **0-1-1-1-0-1-1-1-1-1**. Toto je provedeno s ohledem na generátor kombinací, který v programu používám. Generátor kombinací dostává na vstupu kombinaci předchozí a generuje na jejím základě kombinaci následující. Ze vzniklé kombinace proto vygeneruje kombinaci **0-1-1-1-1-0-0-0-0-0**. Jak je vidět, došlo k přeskočení všech kombinací s daným kořenem a program se přesunul ke kořenu následujícímu.

Identifikace kombinace, kterou je možné odříznout, provádíme dvěma způsoby:

1. Odřezávání zdola podle ceny
2. Odřezávání shora podle váhy

V prvním případě spočítáme cenu hranice, na které se právě nacházíme. Tedy zjistíme, jak nám mohou zbývající předměty zlepšit současný stav za předpokladu, že by je bylo možné použít všechny (nezávisle na jejich váze). Pakliže součet tohoto zlepšení se současnou cenou není lepší, než dosavadní nejlepší řešení, nemá smysl tuto větev řešit dále.

V druhém případě pouze ověříme, zda současné předměty již batoh nepřetěžuj. Pakliže ano, logicky jakýkoliv další předmět může batoh pouze ještě více přetížit.

3.2 METODA DYNAMICKÉHO PROGRAMOVÁNÍ

Základem řešení metodou dynamického programování je dvourozměrné pole, které tvoří tabulku. V této tabulce jsou na jedné ose jednotlivé předměty a na druhé všechny přípustné ceny batohu (horní hranice této osy je tedy součet všech cen předmětů dané instance problému). Obsahem tabulky jsou pak váhy.

O vyřešení problému metodou dynamického programování se stará třída **DynamicProgramming-KnapsackSolver**. Skládá se ze dvou částí a sice:

1. Vyčíslení tabulky
2. Backtracking řešení

Vyčíslení tabulky začíná na pozici [0][0], odkud metoda **enumerateFromPosition** vypočte dva možné následující kroky.

1. Následující předmět se nepoužije
2. Následující předmět se použije

V prvním případě pouze opíšeme hodnotu (váhu) do buňky napravo z té současné.

V druhém případě se také posuneme na ose x o políčko doprava, ale zároveň se posuneme i na ose cen a to o hodnotu nově přidávaného předmětu. Váha bude nastavena na hodnotu výchozí buňky + váha nově přidávaného předmětu.

Po tom, co vypočteme tyto dvě možnosti pro následující předmět, rekurzivně zavoláme tuto metodu znova pro dvě nově vyplněné pozice (tedy metoda vždy rekurzivně volá sebe sama dvakrát, pokud jsou tyto pozice různé). Výjimkou je, pokud současná váha předmětů již překračuje maximální váhu batohu, v tom případě již větev přidávání dalšího předmětu nevoláme pro ušetření času.

3.2.1 Kolize

Problém nastává, pokud se do nějaké buňky tabulky můžeme dostat více cestami (např. součet cen předmětů 1 a 3 je stejná jako předmětů 1 a 5). V tomto případě musíme porovnat váhy jednotlivých řešení a zvolit tu nižší.

3.3 METODA FPTAS ALGORITMU

Tato metoda (Polynomial-time approximation scheme) je pouze modifikací metody dynamického programování, a proto je i v kódu mého programu použita stejná řešící třída, tedy **DynamicProgramming –KnapsackSolver**. Třída byla pouze rozšířena o nový konstruktor, přijímající hodnotu relativní chyby, jakou je klient ochotný akceptovat.

Metoda FPTAS řeší problém psudopolynomiálnosti metody dynamického programování – tedy to, že je složitost závislá nejen na velikosti instance, ale také je citlivá na data (velikosti cen). FPTAS metoda nám dává polynomiálně složitý algoritmus, který je závislý na relativní chybě, kterou jsme ochotni akceptovat.

Implementaci je možné vidět v metodě **getAmountOfBitsByRelativeError**. Tato metoda spočte ze zadané relativní chyby počet bitů, které je možné zanedbat dle vzorce $\mathbf{b = [\log_2(\varepsilon . C_{max} / n)]}$. Tento počet bitů je poté použit následujícím způsobem na všechny ceny předmětů: $cena = cena / 2^b$. Tímto dosáhneme toho, že s každým zvýšením počtu zanedbávaných bitů dojde k dvojnásobnému zmenšení ceny. Např. $400 / 2^1 =$ **200** a $400 / 2^2 =$ **100**. Samotný průběh algoritmu je zcela stejný, jako když problém řešíme pomocí dynamického programování bez snižování cen.

Důsledkem snížení ceny předmětů je snížení paměťové a tedy i časová náročnosti (jeden rozměr tabulky dynamického programování je dán součtem všech cen předmětů).

4 NAMĚŘENÉ VÝSLEDKY

4.1 HW / SW KONFIGURACE TESTOVACÍHO SYSTÉMU

- CPU Intel® Core™2 Duo; 2.26 GHz, 2.27 GHz
- 4 GB RAM
- OS Windows 8 64-bit
- Java 7

4.2 ZPŮSOB MĚŘENÍ

Čas byl měřen pomocí třídy **ThreadMXBean** a byl průměrován přes všechny instance dané velikosti. Pro úlohy příliš malé velikosti bylo měření spuštěno v cyklu vícekrát a následně vyděleno počtem běhů.

Chyby cen FPTAS aproximaci oproti hrubé síly jsem počítal dle vzorce ε = (C(OPT) - C(APX)) / C(OPT) a chyby byly počítány průběžně, ne na finálním součtu cen.

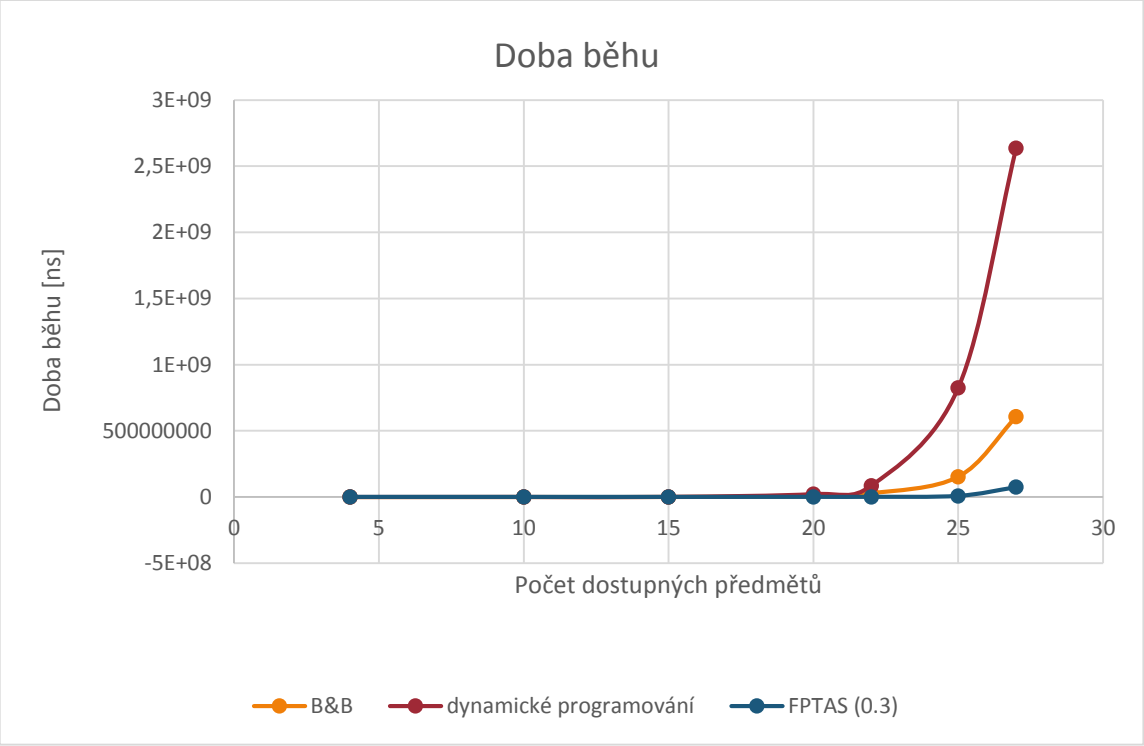
4.3 VÝSLEDKY

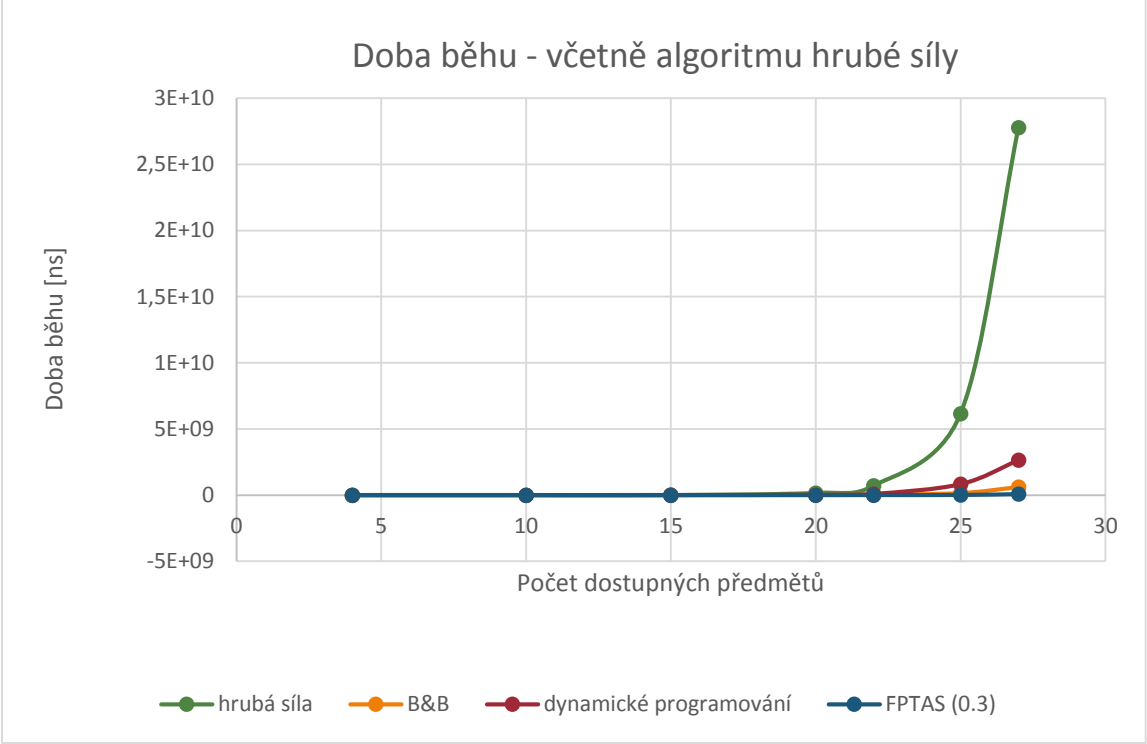
Počet předmětů	Hrubá síla [ns]	B&B [ns]	dynamické programování [ns]	FPTAS (ε = 0,3) [ns]
4	1025	312002	6084	5740
10	101899	936006	46488	29744
15	4368028	1248008	730084	62790
20	168574680	7176046	21902540	187201
22	720100616	29016186	85800550	530403
25	6159543484	153816986	824621286	7987251
27	27779098070	606843890	2637664908	74100475

Pro lepší názornost jsem vytvořil dva grafy. Jeden bez hrubé síly a jeden s ní, protože doby běhu těchto algoritmů se velmi liší.

4.3.1 Čas běhu metody větví a hranice (B&B)

Metoda větví a hranic je sice úplná a její asymptotická složitost je stejně jako u hrubé síly $O(2^n)$, ale jak je vidět, reálně je B&B mnohem rychlejší. Složitost se nedá exaktně určit, protože je závislá na konkrétních datech (s jednou instancí se nám může podařit odříznout většinu stavového prostoru malým počtem řezů, s jinou se nám nemusí podařit odříznout nic).


Doba běhu
Doba běhu [ns]
3E+09
2,5E+09
2E+09
1,5E+09
1E+09
500000000
0
-5E+08
0
5
10
15
20
25
30
Počet dostupných předmětů
B&B
dynamické programování
FPTAS (0.3)


Doba běhu - včetně algoritmu hrubé síly
Doba běhu [ns]
3E+10
2,5E+10
2E+10
1,5E+10
1E+10
5E+09
0
-5E+09
0
5
10
15
20
25
30
Počet dostupných předmětů
hrubá síla
B&B
dynamické programování
FPTAS (0.3)

4.3.2 Čas běhu metody dynamického programování

Metoda dynamického programování má asymptotickou složitost $O(n^2*C_{max})$, kde C_{max} je součet všech cen předmětů a n počet předmětů k dispozici. Toto znamená, že je algoritmus polynomiální vzhledem k délce vstupu, nikoliv však k velikosti vstupu. Problém nastane, pokud dostaneme zadané vysoké ceny nebo v nediskrétní množině. Tento problém řeší následující metoda.

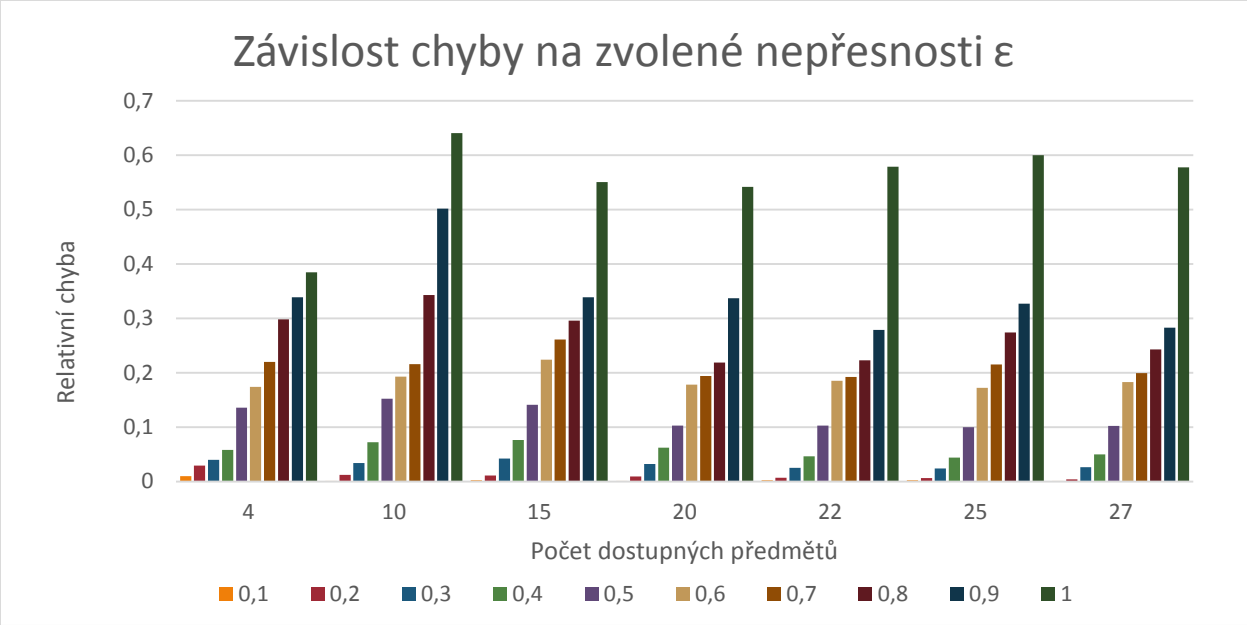
4.3.3 Čas běhu metody FPTAS

Metoda FPTAS řeší problém možného růstu složitosti do nekonečna s cenami předmětů. Řešením je zanedbání určitého počtu bitů cen, čímž omezíme maximální možnou velikost ceny. Ztratíme tím však přesnost algoritmu, a proto je vhodné mít možnost definovat maximální povolenou chybu. Pokud použijeme pro výpočet počtu bitů k zanedbání vzorec $b = \lceil \log_2(\varepsilon*C_{max} / n) \rceil$, který jsme odvodili ze vzorce pro výpočet relativní chyby $\varepsilon = n*2^b / C_{max}$, dostaneme se ze složitosti metody dynamického programování $O(n^2*C_{max})$ na složitost $O(n^3/\varepsilon)$, kde již není zahrnuta instanční hodnota C_{max}. Algoritmus je tedy plně polynomiální. Jak je vidět na grafu, při použití $\varepsilon = 0{,}3$ je algoritmus rychlejší i než B&B.

4.3.4 Závislost chyby na zvolené nepřesnosti ε

relativní chyba	**povolená nepřesnost ε**									
Počet dostupných předmětů	0,1	0,2	0,3	0,4	0,5	0,6	0,7	0,8	0,9	1
4	0,01	0,029	0,04	0,058	0,136	0,174	0,22	0,298	0,339	0,385
10	0,001	0,012	0,034	0,072	0,152	0,193	0,216	0,343	0,502	0,641
15	0,002	0,011	0,042	0,076	0,141	0,224	0,261	0,296	0,339	0,551
20	0,001	0,009	0,032	0,062	0,103	0,178	0,194	0,219	0,337	0,542
22	0,002	0,007	0,025	0,046	0,103	0,185	0,192	0,223	0,279	0,579
25	0,002	0,006	0,024	0,044	0,1	0,172	0,215	0,274	0,327	0,6
27	0,001	0,004	0,026	0,05	0,102	0,183	0,199	0,243	0,283	0,578

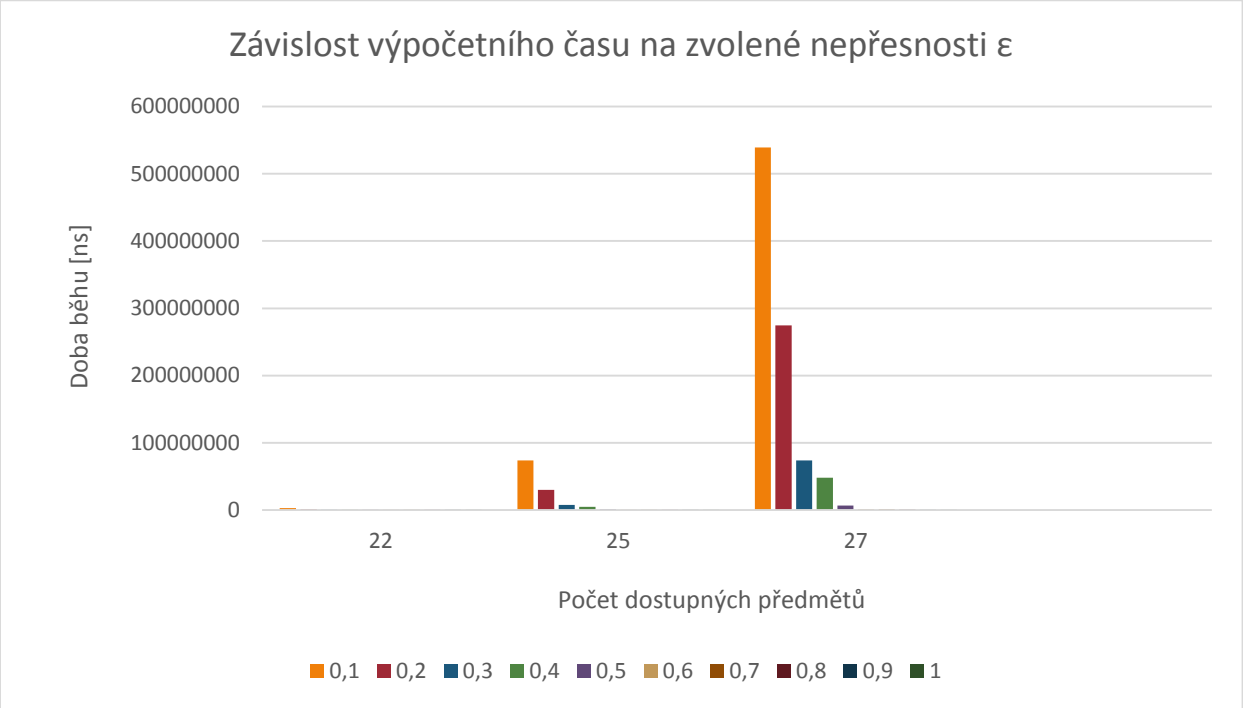
Je vidět, že chyba je přímo úměrná (polynomiálně) nastavené nepřesnosti ε.



4.3.5 Závislost výpočetního času na zvolené nepřesnosti ε

Doba běhu [ns]	**povolená nepřesnost ε**									
Počet dostupných předmětů	**0,1**	**0,2**	**0,3**	**0,4**	**0,5**	**0,6**	**0,7**	**0,8**	**0,9**	**1**
4	5865	5772	5740	5725	5709	5647	5616	5599	5584	5553
10	29952	29865	29744	29640	29535	29432	29222	29120	29016	28704
15	63960	63123	62790	62400	61620	61230	60840	59670	59435	58110
20	399362	299521	187201	143520	137280	124800	118560	118454	112499	112320
22	2979619	1622410	530403	421202	280801	218401	187201	171601	156001	140400
25	73694872	30326594	7987251	4867231	1372808	436802	374402	333422	312002	249601
27	538983455	274561760	74100475	48048308	6708043	1560010	1404009	1248008	624004	780005

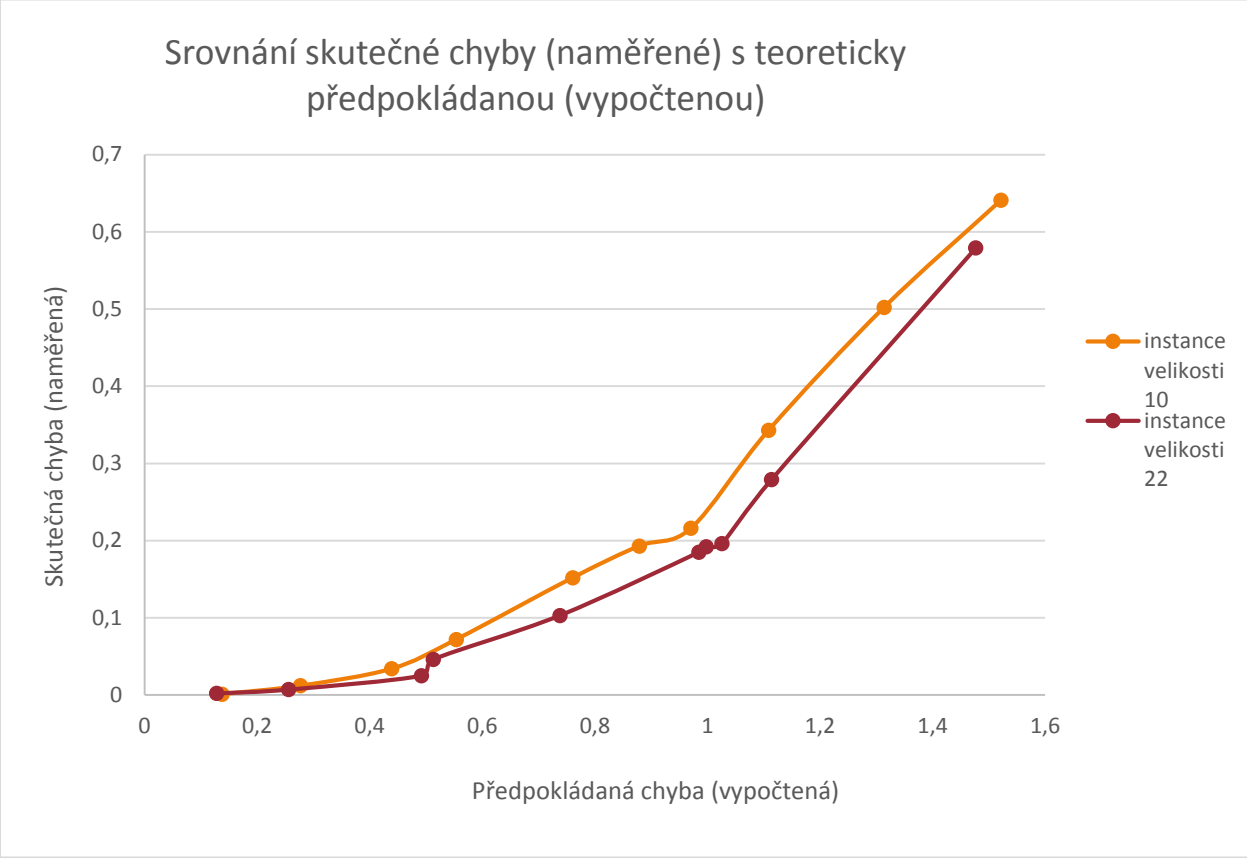
Pro lepší názornost jsem do grafu zahrnul pouze tři různé velikosti instancí. Z naměřených hodnot a grafu je vidět, že se zvýšením povolené nepřesnosti ε o 0,1 dojde přibližně k dvojnásobnému zrychlení výpočtu.



4.3.6 Srovnání skutečné chyby (naměřené) s teoreticky předpokládanou (vypočtenou)

	instance velikosti 10			instance velikosti 22		
povolená nepřesnost ε	předpokládaná chyba	skutečná chyba	zanedbaných bitů (vypočteno na základě ε)	předpokládaná chyba	skutečná chyba	zanedbaných bitů (vypočteno na základě ε)
0,1	0,138	0,001	4	0,128	0,002	3
0,2	0,277	0,012	5	0,256	0,007	4
0,3	0,439	0,034	5	0,492	0,025	5
0,4	0,554	0,072	6	0,513	0,046	5
0,5	0,761	0,152	6	0,738	0,103	6
0,6	0,879	0,193	6	0,985	0,185	6
0,7	0,971	0,216	6	0,998	0,192	6
0,8	1,109	0,343	7	1,026	0,196	6
0,9	1,314	0,502	7	1,114	0,279	7
1	1,522	0,641	7	1,477	0,579	7

Jak je vidět, poměr skutečné chyby k vypočtené (dle vzorce $\varepsilon = n*2^b / C_{max}$) je přibližně stejný jak pro velké, tak pro malé instance. Skutečná chyba je mnohem menší, než ta předpokládaná.



5 ZÁVĚR

Ačkoliv teorie naznačuje něco jiného, na základě měření je v případě problému batohu metoda **B&B rychlejší, než dynamické programování**. To se stane rychlejším jen při kombinaci s metodou FPTAS a smířením se s určitou úrovní relativní chyby.

Několik zjištění stran metody FPTAS a nastavené nepřesnosti ε:

1. FPTAS dělá z pseudopolynomiální složitosti dynamického programování plně polynomiální.
2. Nastavení větší přesnosti (snížení ε) FPTAS zvýší dobu výpočtu, avšak tato závislost je nanejvýš polynomiální (dle měření jde o dvojnásobek). Toto se děje proto, že s každým dalším zanedbaným bitem dojde k zmenšení stavového prostoru o polovinu oproti předchozímu stavu.
3. FPTAS je silným nástrojem především tehdy, když potřebujeme diskretizovat optimalizační kritérium (cena s desetinou čárkou).
4. I malé množství zanedbaných bitů může velmi urychlit výpočet při zachování dobré kvality výsledků. Např. při zanedbání 5 bitů (ε=0,3 u instancí velikosti 22 s danými cenami) došlo k relativní chybě pouze 0,025, avšak zrychlení výpočtu je přibližně 170x.