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

Zadání úlohy:

- Naprogramujte řešení problému batohu. Na zkušebních datech pozorujte závislost výpočetního času na n. U algoritmu FPTAS pozorujte závislost výstupní chyby a času na zvolené relativní chybě.
- Použité metody:
  - ↘ Metoda hrubé síly
  - ↘ Metoda větví a hranic
  - ↘ Metoda dynamického programování
  - ↘ Metoda FPTAS

Rozbor řešení:

Výpočet pomocí hrubé síly jsem převzal z úlohy č. 1 a její popis je zahrnut v předchozí zprávě.

Metoda větví a hranic je rozšíření naivního přístupu o horní a spodní hranice. Jedna hranice je určena celkovou přidanou váhou a kapacitou batohu, druhá srovnává nejlepší nalezené řešení a nejlepší možné řešení současného podstromu řešení.

Algoritmus dynamického je pseudopolynomiální algoritmus, který vyplňuje 2D tabulku, tak aby vyzkoušel vhodné kombinace. Složitost je proto závislá na počtu předmětů a celková ceně s horní mezí $O(m * n)$, kde n je počet předmětů a m je jejich celková cena. V implementaci jsem použil rozklad podle ceny, protože na jeho základě je implementován i algoritmus FPTAS.

Metoda FPTAS je plně polynomiální heuristika, která umožňuje nalezení řešení batohu s volitelnou přesností. Pomocí parametru ε můžeme určit maximální chybu. Platí, že se zvyšováním přesnosti neporoste výpočetní složitost více jak polynomiálně.

Experimentální vyhodnocení:

Měření jsem prováděl pro větší počet opakování. Výsledky vynesené v grafech jsou průměry výsledných časů. Proto i hodnoty, které jsou na hranici měřitelnosti, dávají smysl.

V předmětu Efektivní implementace algoritmů jsme si vyzkoušeli implementaci několika algoritmů pro práci s desetinnými čísly. Použil jsem proto již vyzkoušené přepínače kompilátorů, které jsem upravil pro tuto úlohu.

Použité přepínače kompilátorů GCC:

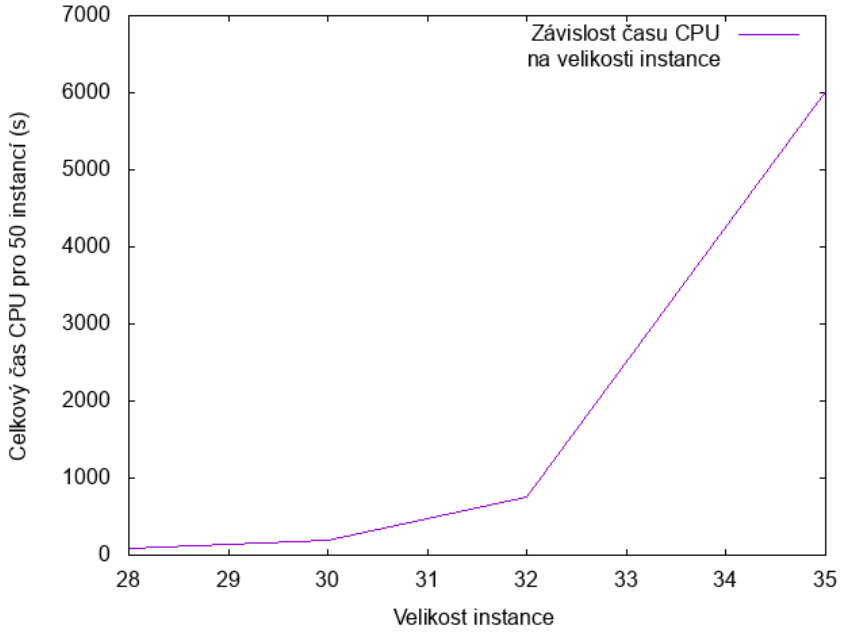
-Ofast -flto -Wno-sign-compare -std=c++11 -fopenmp -fstrict-aliasing -mfpmath=sse -mavx -mpc32 -funsafe-math-optimizations -ffast-math

Testovací soustava:

Intel(R) Core(TM) i5-3570K CPU @ 3.40GHz + 4 cores + Hyper Threading

Úlohy byly spouštěny v režimu jednoho vlákna a na PC neběžela žádná náročná úloha. K měření času jsem použil standardní knihovnu, se kterou mám dobré zkušenosti.

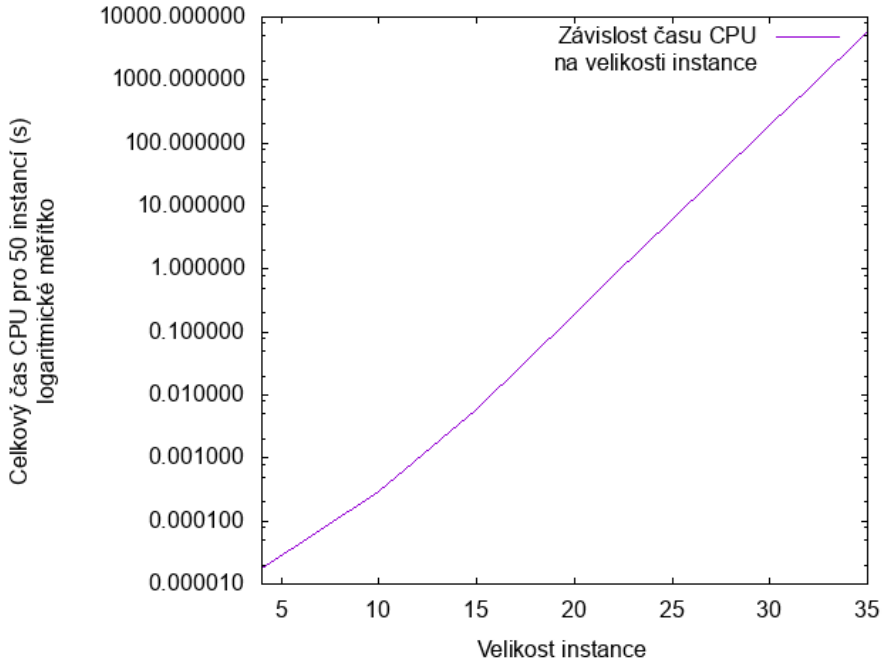
Naivní bruteforce řešení:



1. Graf závislosti celkového času (s) pro 50 instancí na její velikosti

Z důvodu nedostatečného výkonu jsem vynechal instance o velikostech 37 a 40, protože řešení by trvalo několik hodin.

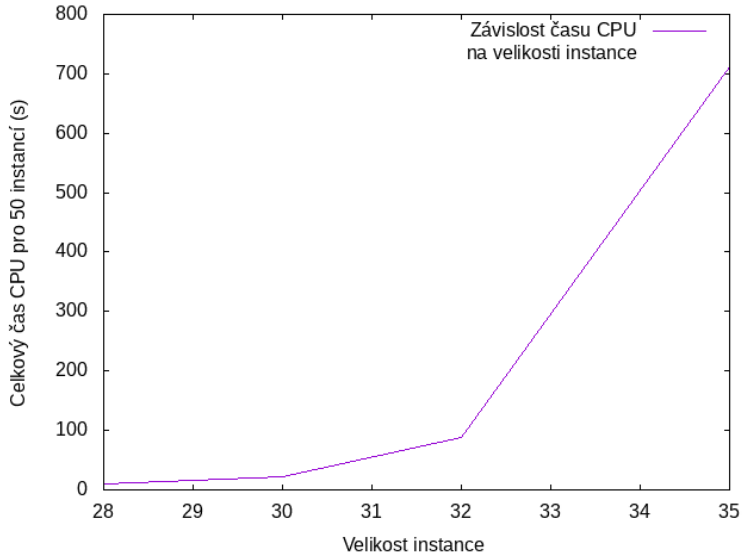
Graf [1] naměřených časů pro naivní řešení kopírují průběh exponenciály, tím podporují odhadovanou exponenciální závislost časové složitosti na velikosti vstupní instance.



2. Graf závislosti celkového času (s) pro 50 instancí na její velikosti s logaritmickým měřítkem na ose y

Z grafu [2] je vidět, že závislost je téměř lineární. Protože na ose y je aplikováno logaritmické měřítko, je závislost v datech exponenciální. Tento výsledek podporuje náš odhad.

Metoda větví a hranic:



3. Graf závislosti celkového času (s) pro 50 instancí na její velikosti

Metoda větví a hranic nemění asymptotickou složitost, která zůstává $O(n^2)$, ale upravuje konstantu. Díky horní a spodní mezi řešení je strom generovaných řešení řidší. Stále je ovšem možné najít instanci problému pro kterou bude i tato metoda procházet velký počet možností a proto jsou časové výsledky poměrně nevyvážené a silně datově citlivé.

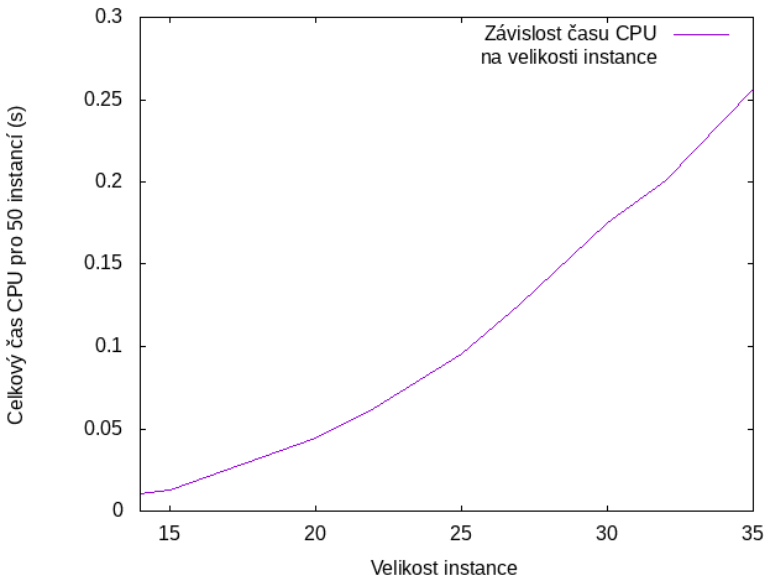
Graf [3], na kterém jsou tyto hodnoty vyneseny, ukazuje, že složitost skutečně zůstává přibližně exponenciální. Přesto došlo k velkému časovému zlepšení oproti naivní metodě.

Výpočet dynamickým programováním:

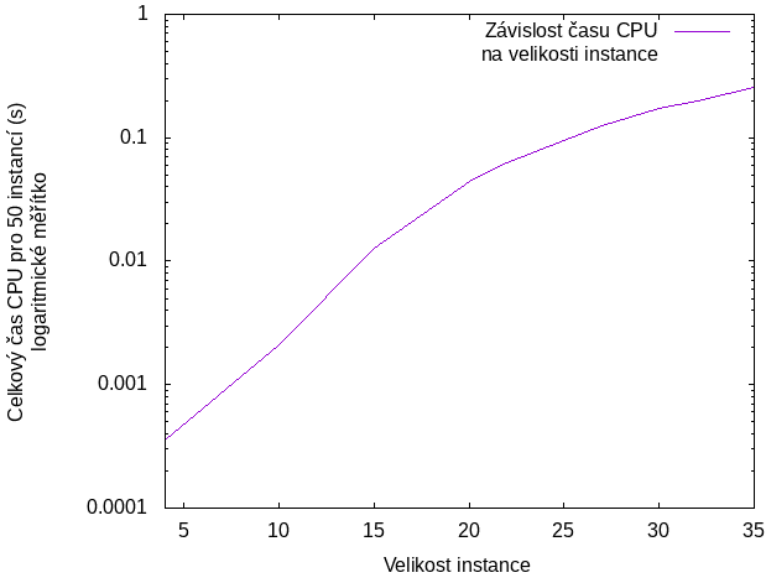
Algoritmus dynamického programování řeší podproblémy, které rekurzivně spojuje. Použitím tabulky eliminuje redundanci ve výpočtu.

Vybral jsem si metodu dekompozice podle ceny, kdy do tabulky ukládáme minimální váhu, za kterou jsme schopni dosáhnout počtu předmětů a dané ceny.

Do celkové složitosti $O(m * n)$, kde m je celková cena a n je počet předmětů, se projevuje i časová náročnost paměťového alokátoru, který musí alokovat $O(m * n)$ jednotek paměti.



4. Graf závislosti celkového času (s) pro 50 instancí na její velikost



5. Graf závislosti cel. času (s) pro 50 instancí na její velikosti s log. měřítkem na ose y

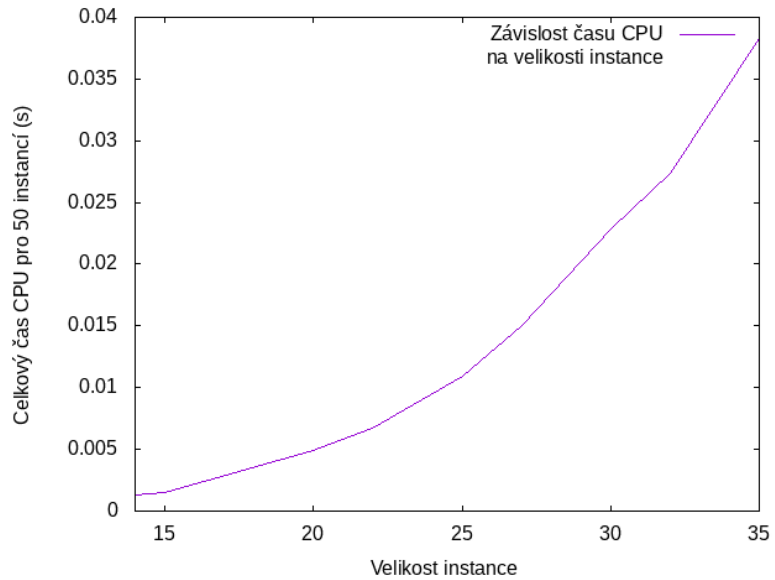
V grafu [5] jsem zvolil na ose y logaritmické měřítko, abych ukázal, že složitost není exponenciální. Toto je z grafu vidět, protože výsledný průběh je sublineární.

Oproti metodě větví a hranic jsou výsledky daleko vyváženější a je obtížnější najít instanci, která výrazně ovlivní časový průběh algoritmu.

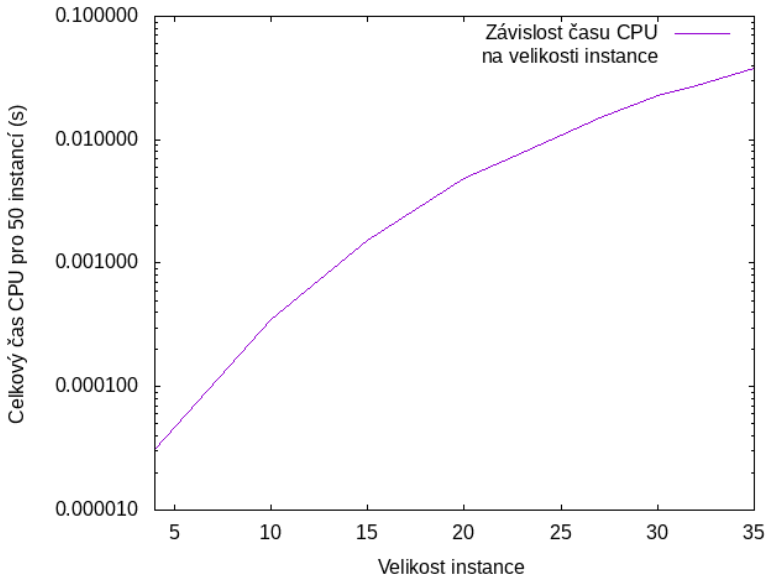
Algoritmus FPTAS:

Metoda FPTAS je aproximační heuristika pro problém batohu v deterministické variantě. Umožňuje nastavení relativní chyby oproti optimálnímu řešení. Tento algoritmus rozšiřuje algoritmus dynamického programování při rozkladu podle ceny.

Velkou výhodou tohoto postupu je možnost volit relativní chybu oproti optimálnímu řešení, kdy máme zaručeno, že chybovost nebude vyšší, než zvolená proměnná.



6. Graf závislosti celkového času (s) pro 50 instancí na její velikost pro ε = 0.9



7. Graf závislosti celkového času (s) pro 50 instancí na její velikosti s log. měřítkem na ose y pro ε = 0.9

V grafu [7] jsem zvolil na ose y logaritmické měřítko, abych ukázal, že složitost není exponenciální. Toto je z grafu vidět, protože výsledný průběh je sublineární.

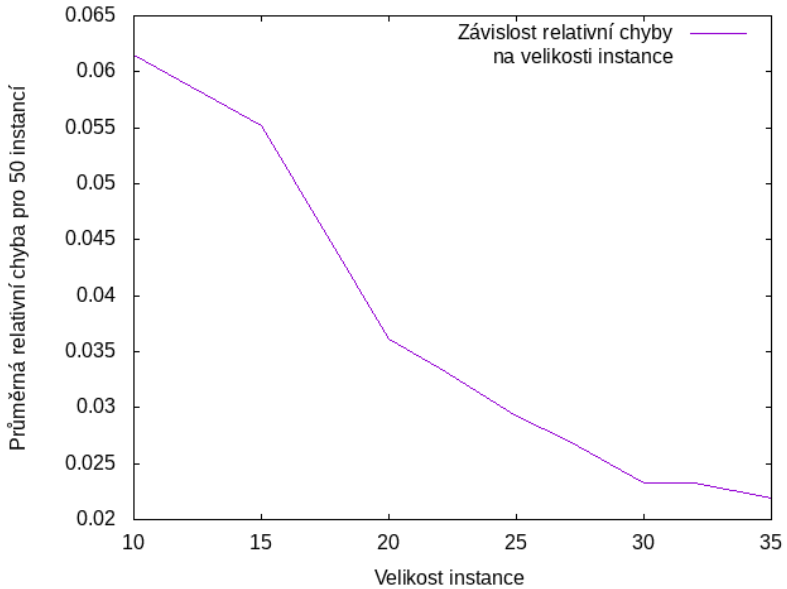
Stejně jako u metody dynamického programování jsou výsledky vyváženější oproti metodě větví a hranic a je obtížnější najít instanci, která výrazně ovlivní časový průběh algoritmu. Výsledný čas naměření pro ε = 0.9 zrychlil výpočet přibližně o jeden řád.

Relativní chyba FPTAS:

Měření relativní chyby jsem prováděl tak, jak je doporučeno v materiálech na eduxu pomocí vzorce a následného zprůměrování pro zadané instance:

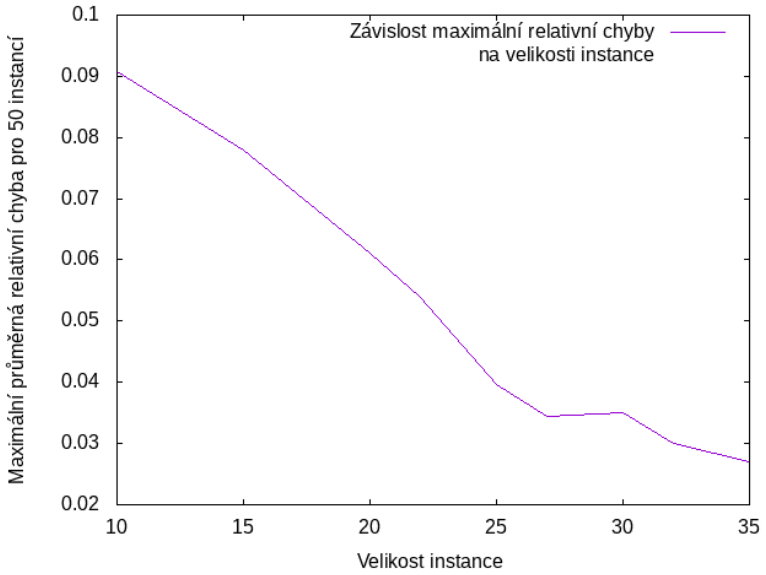
$$\varepsilon_{rel} = \frac{C(OPT) - C(APX)}{C(OPT)}$$

$C(OPT)$ je cena opt. řešení
$C(APX)$ je cena vrácená heuristickou metodou.



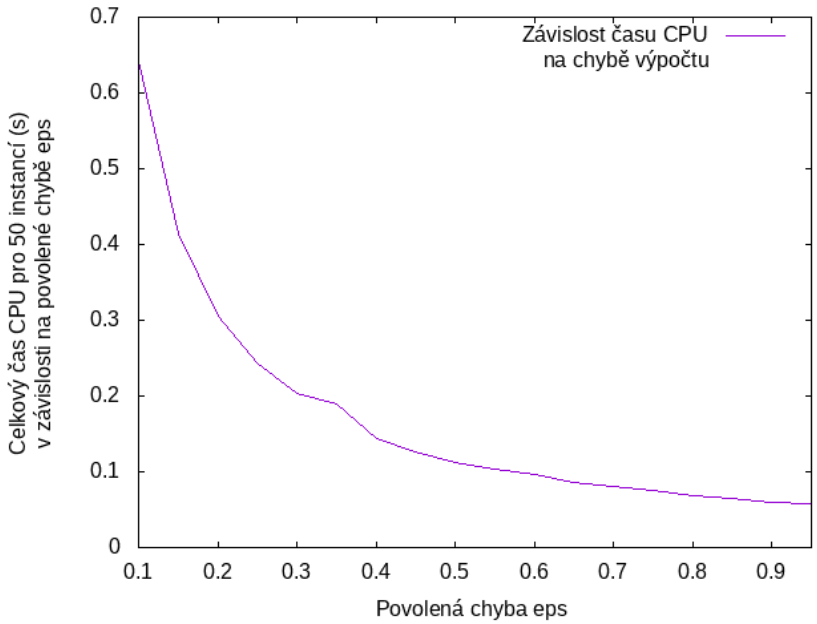
8. Graf závislosti průměrné relativní chyby pro 50 instancí na její velikosti pro ε_{max} = 0.9.

Graf [8] zobrazuje průměrnou relativní chybu, která porovnává cenu nejlepšího řešení pro 50 instancí problému. Z grafu je čitelné, že chyba klesá se zvětšující se velikostí instance.



9. Graf závislosti maximální relativní chyby pro 50 instancí na její velikosti pro ε_{max} = 0.9.

Graf [9] zobrazuje maximální relativní chybu, která porovnává cenu nejlepšího řešení pro 50 instancí problému. Z grafu je čitelné, že maximální chyba nikdy nepřekročila zadanou hranici přesnosti ε_{max}= 0.9.



10. Graf závislosti celkového času CPU pro 50 instancí na velikosti povolené chyby.

Graf [10] zobrazuje celkový čas CPU v závislosti pro zvolené maximální chybě. Je vidět, že celkový spotřebovaný čas klesá pro vyšší přípustnou chybu.

Závěr:

Během měření jsme zjistili, že rozdíl exaktní metody a metody větví a hranic je poměrně vysoký, ale záleží na konkrétní instanci. Můžeme říct, že metoda BB se bude v nejhorším případě chovat jako naivní řešení.

Algoritmus dynamického programování mění horní mez složitosti, protože jeho složitost nezávisí exponenciálně na počtu předmětů, ale je ovlivněna hodnotami dané instance. Proto je důležité prozkoumat instance, které chceme řešit, a vybrat vhodný algoritmus. Poté vycházíme z nových informací a můžeme se tomu přizpůsobit.

Zajímavý je algoritmus FPTAS, který umožňuje volit maximální chybu řešení. Změna přesnosti ovlivňuje výsledný čas, a proto je nutné určit naše požadavky a vyvážit požadovanou přesnost a čas, který chceme za výpočet zaplatit.