2. úkol MI-PAA

Jan Jůna (junajan)
3.11.2013

Specifikaci úlohy

Problém batohu je jedním z nejjednodušších NP-těžkých problémů. V literatuře najdeme množství jeho variant, které mají obecně různé nároky na algoritmus řešení.

Je dáno:

- celé číslo *n* (počet věcí)
- celé číslo *M* (kapacita batohu)
- konečná množina $V = \{v_1, v_2, \dots, v_n\}$ (hmotnosti věcí)
- konečná množina $C = \{c_1, c_2, \dots, c_n\}$ (ceny věcí)

V našem řešení se omezíme na verzi „0/1 problém batohu“, tedy zkonstruujeme takovou množinu $X=\{x_1, x_2, \dots, x_n\}$, kde každé x_i je 0 nebo 1, tak, aby platilo:

$v_1x_1 + v_2x_2 + \dots + v_nx_n <= M$ (aby batoh nebyl přetížen).

a výraz

$c_1x_1 + c_2x_2 + \dots + c_nx_n$

nabýval maximální hodnoty pro všechny takové množiny (cena věcí v batohu byla maximální).

Více viz. https://edux.fit.cvut.cz/courses/MI-PAA/tutorials/batoh

Rozbor variant řešení

Oproti prvnímu úkolu, kdy byl stejný problém řešený metodami **hrubé síly** a **jednoduché heuristiky** podle poměru ceny a váhy, jsme řešili problém 3mi novými způsoby:

- **Metodou větví a hranic** (B&B), která je velmi podobná k řešení hrubé síly s tím rozdílem, že se při procházení stavového prostoru bere ohled na aktuální váhu věcí v batohu. Pokud tato váha přesáhne maximální možnou váhu, je výpočet ukončen a předán zpět do předchozí větve rekurze. Dalším kritériem je ořezávání stavového prostoru podle nejlepší dosažené ceny. Do rekurze se předává nejlepší dosažená cena, nad kterou již výpočet dalších větví rekurze nemá cenu a opět se ukončuje a vrací zpět.
- **Dynamické programování** neboli dekompozice podle kapacity nebo podle ceny je další z možných řešení daného problému. Popis řešení dekompozice podle ceny je popsán například zde: http://www.algoritmy.net/article/5521/Batoh
- **FPTAS algoritmus** tj. s použitím modifikovaného dynamického programování s dekompozicí podle **ceny**. Rozdílem mezi předchozí verzí je pak zvolení maximální chyby, která může dojít

při výpočtu a následného upravení pole vstupních dat.

Popis kostry algoritmu

Implementace všech popsaných algoritmů byla provedena v programovacím jazyce Python verze 3.2. Společné funkce algoritmů byly uloženy do souboru common.py, který je importován níže popsaných algoritmů.

Algoritmus pro výpočet metodou větví a hranic

Algoritmus (**bb.py**) nejprve načte podle parametrů název souboru se vstupními daty a případně i počet opakování, kolikrát se má výpočet provést. Poté jsou načtena vstupní data, ty jsou rozpársována a serializována pro další použití. Další zpracování přejímá funkce *processProblem* která vytvoří počáteční instanci výsledku, do které se dále bude doplňovat nejlepší řešení a nulový vektor o velikosti vstupních dat.

Rekurzivní funkce *iterateProblem* přejímá vstupní data a vektor, tvořený z jedniček a nul. Zařazení prvku do batohu pak určuje jednička na stejné pozici, jakou má prvek ve vstupních datech. Funkce vždy ověří, zda vnoření již nepřekročuje velikost vstupních dat. Pokud ne, otestuje zda současná konfigurace vektoru nepřesahuje maximální velikost baťohu. V takovém případě vrací aktuální nejlepší současnou konfiguraci. Poté ověří, zda konfigurace vstupního vektoru nemá lepší cenu než současná nejlepší, pokud ano, uloží ji a pokračuje na další vnořování. V předávaných datech je také informace o nejlepší dosažené ceně. Tato hodnota hraje velkou roli ve zpracovávání rekurze, kdy se při rekurzivním postupu ve stavovém prostoru vždy vezmou všechny prvky zbývající k prozkoumání, spočítá se jejich cena a pokud je tato cena + aktuálně dosažená menší jak zatím nejlepší dosažená, je zbytečné tuto větev stavového prostoru procházet, protože nám nemůže poskytnout nové nejlepší řešení.

Pokud rekurze projde všechny permutace problému – ve vstupním vektoru se změnily všechny konfigurační bity – vrátí se nejlepší naměřený výsledek uložený v proměnné *BEST_PART*.

Pomocí druhého nepovinného parametru programu se dá určit počet opakování výpočtu problému na jednu instanci. Tento počet se dá použít především u krátkých výpočtů, u kterých by byl problém s měřením času. Instance je tak spočítána vícekrát a naměřený čas je podělen počtem opakování

Na **stdout** se vytiskne se výsledek a do **stderr** se vloží ID záznamu, čas celého opakovaného řešení instance, počet opakování a čas řešení instance (celkový čas na instanci / počet opakování). Poté se přejde k další instanci problému.

Algoritmus pro výpočet dynamickým programováním

Algoritmus pro výpočet zadanou heuristikou **(dynamic.py)** začíná stejně jako v prvním případě a to načtením vstupních dat, jejich rozparsováním.

Oproti prvnímu případu však prochází zadanými vstupní data a sestavuje **2D pole**, jehož jedna strana uvádí dosaženou cenu věcí v baťohu, druhá je pak Xtá prozkoumaná položka. Hodnota v 2D poli pak určuje dosaženou váhu zvolených předmětů.

Krok algoritmu je tedy následující: Mohou nastat dvě situace – dojde proto k rozdělení řešení a algoritmus bude postupovat po obou větvích. První možností je, že algoritmus do batohu přidá danou položku a přejde na souřadnici **[y+1;x+cena předmětu]**, kde x je dosavadní součet cen položek

obsažených v batohu a nastaví hodnotu pole na **z + váha předmětu** (z je dosavadní součet vah všech předmětů obsažených v batohu). Druhou možností je, že předmět do batohu nebude přidán – algoritmus přejde na souřadnici **[y+1;x]**, kde hodnotu pole stanoví opět jako součet vah všech obsažených předmětů (protože nebyl přidán žádný předmět, tak hodnotu pouze opíše ze zdrojového pole). Stejným způsobem algoritmus postupuje pro všechny dosažené buňky v následujícím řádku, dokud nejsou zpracovány všechny předměty (vyplněny všechny řádky).

Řešením je pak taková buňka, jejíž hodnota (váha věcí) je maximální možná tak, aby nepřekročila maximální váhu baťohu a zároveň Ynová souřadnice (cena) je největší možná.

Zpětnou podobu vektoru pak dostaneme zpětným průchodem spočítané tabulky tak, že vždy odečítáme cenu předmětu (dostaneme novou Ynovou souřadnici) a váhu předmětu (nová váha – hodnota buňky) a dekrementujeme Ynovou souřadnici o 1. Pokud je v tabulce na Ynové souřadnici o 1 menší stejná váha jako je váha aktuální zpracované buňky, znamená to, že v konečném řešení prvek není, tudíž dekrementujeme Ynovou souřadnici a pokračujeme dál.

Popis problému a jeho řešení pomocí dynamického programování je popsán a graficky znázorněn například zde: http://www.algoritmy.net/article/5521/Batoh

Algoritmus FPTAS

Algoritmus (**fptas.py**) je velmi podobný řešení pomocí dynamického programování s dekompozicí podle ceny. Rozdílem je pak akorát prvotní předzpracování pole, kdy se ceny předmětů zmenší o nějaký faktor K, který je spočítán jako zvolená maximální procentuelní chyba * největší cena / počet předmětů.

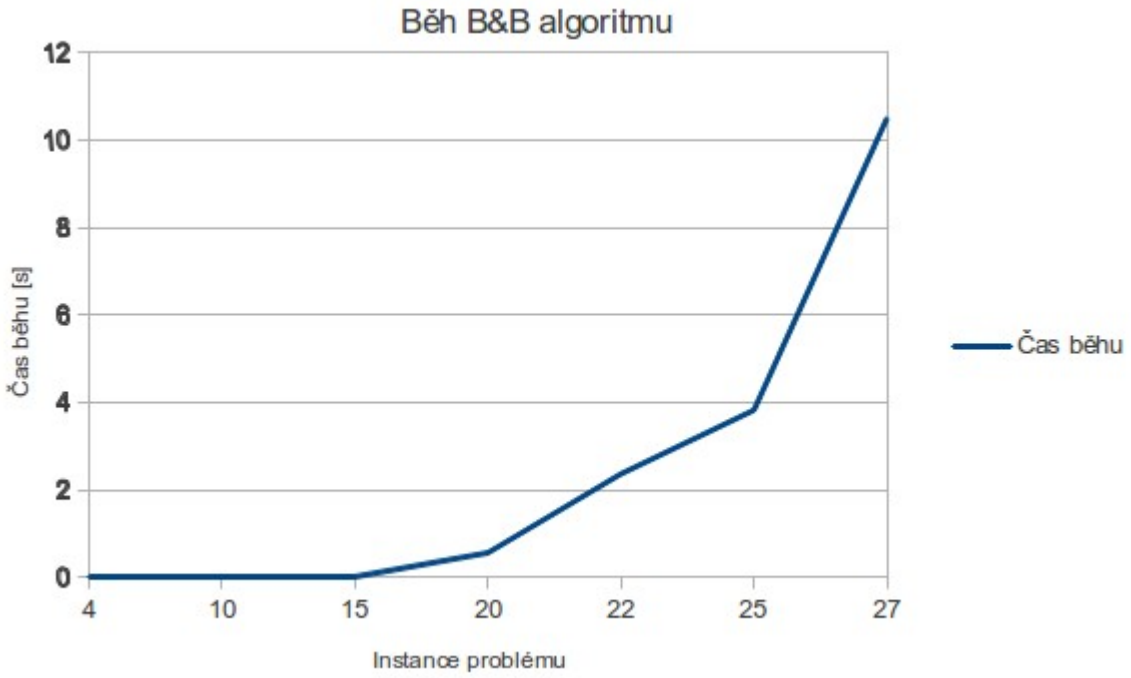
Algoritmus s dekompozicí podle ceny poté spočítá maximálni možnou cenu v baťohu, o které můžeme říct, že nabývá hodnot se zvolenou maximáln procentuelní odchylkou.

Naměřené výsledky

Algoritmus byl testován na počítači s procesorem Intel® Core™ i5-2540MSandy Bridge.

Algoritmus pro výpočet metodou větví a hranic

Pro měření prvního algoritmu byly použity vstupní data o velikosti 4,10,15,20,22,25, 25 a 27 prvků. Časová složitost algoritmu na těchto datech je zobrazena v grafu níže.



Data jsou také zobrazena v následující tabulce:

	4	10	15	20	22	25	27
Čas běhu [s]	0,000148	0,00322	0,021567	0,556	2,361	3,8208	10,496842

Podle grafu je vidět, že čas na výpočet těchto dat rostě exponenciálně vzhledem k velikosti vstupní množiny.

Algoritmus pro výpočet dynamickým programováním

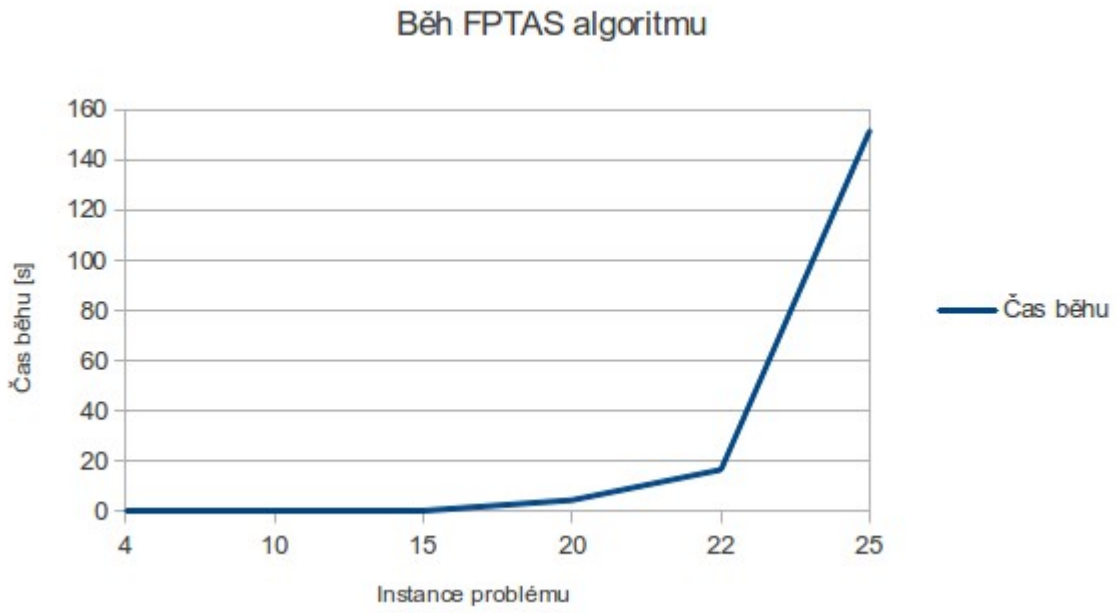
Výpočet byl opět otestován na vstupních datech o velikosti 4 až 25 vstupních prvků.

Data jsou také zobrazena v následující tabulce:

	4	10	15	20	22	25
Čas běhu [s]	0,000032	0,003126	0,16488	5,6542	21,013	181,96

FTPAS algoritmus

Vývoj časové složitosti FTPAS algoritmu na datech od 4 do 25 položek:



Data jsou zobrazena také v následující tabulce:

	4	10	15	20	22	25
Čas běhu [s]	0,000027	0,00162	0,09232	4,5244	16,6288	151,823

Pro řešení algoritmem FPTAS je nutné vzít v úvahu také možnou chybu, při které může při výpočtu dojít. Algoritmus pracuje s hodnotou **epsilon** u nás nastavenou na 10%, která říka, jaká může být maximální odchylka od nejlepšího výsledku.

Relativní chyba se pro maximalizační problémy počítá jako rozdíl ceny optima a ceny přibližného řešení podělený opět cenou přibližného řešení. Tedy: $\varepsilon = (C(OPT)-C(APX)) / C(OPT)$

- kde C(OPT) je cena optima
- a C(APX) je cena přibližného řešení

Naměřené odchylky jsou uvedeny v následujcí tabulce:

Počet prvků	**Průměrná odchylka**	**Maximální odchylka**
4	4,140399	5,456522
10	1,346641	1,579882

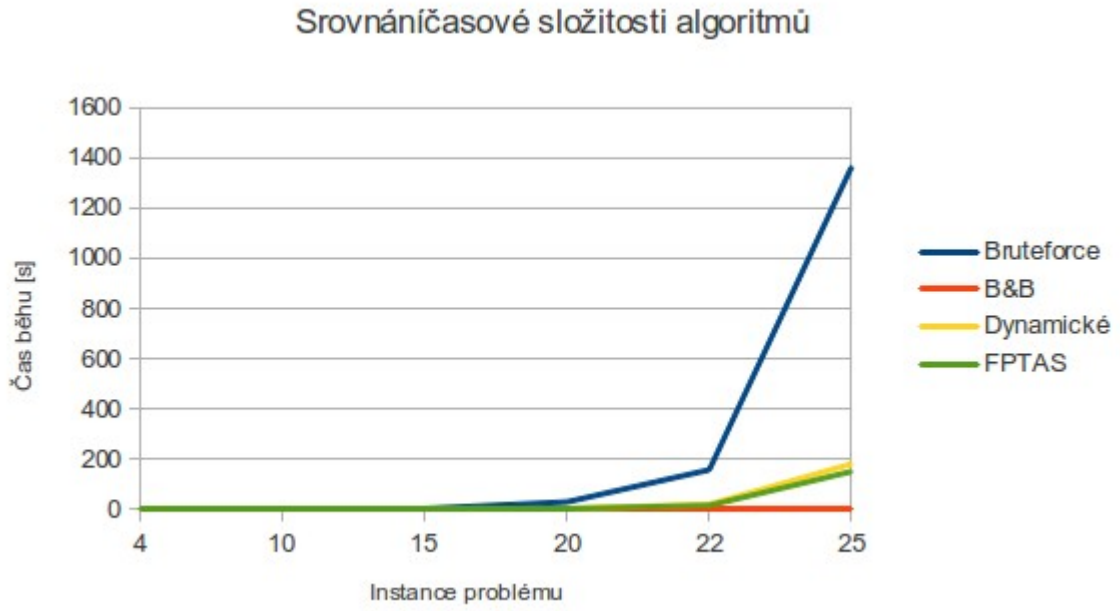
15	0,607584	0,707458
20	0,213884	0,283460
22	0,115084	0,164322
25	0,012954	0,022695

Maximální naměřená odchylka je velikostí rovna hodnotě 5,456522.

Pro spočítání odchylky byl vytvořen script **countDiv.py**. Průměr časové složitosti je počítán pomocí scriptu **countTime.py**

Srovnání časové složitosti algoritmů

Pro představu je přiložen i graf se srovnáním výše zmíněných 3 algoritmů spolu s bruteforce algoritmem z minulého úkolu.



Závěr

Z naměřených hodnot u jednotlivých instancí řešených pomocí B&B algoritmu vyplývá, že je řešení závislé na pozici problému v řešeném stavovém prostoru. Pokud nalezneme nejlepší možné řešení hned na začátku prohledávání, je stavový prostor velmi rychle ořezán a řešení je tak zrychleno až několika násobně. Pokud nejlepší řešení leží až na konci stavového prostoru (nejhorší případ), má

řešení složitost stejnou jako řešení pomocí brute force algoritmu.

Všechny zde uvedené algoritmy jsou velmi závislé na velikosti vstupních dat, jejich časová složitost je tak v nejhorším případě exponenciální.

FPTAS algoritmus také počítá s jistou potencionální chybou. Rychlost zpracování ovšem vychází o trochu lépe, jak jeho předchůdce – dynamické programování.

Zdroje

Popis problému baťohu: https://edux.fit.cvut.cz/courses/MI-PAA/tutorials/batoh

Dynamické programování: http://www.algoritmy.net/article/5521/Batoh

Zdrojové kódy: http://honza.dreamware.cz/Batoh2.zip