NI-KOP – úkol 2

Ondřej Kvapil

Kombinatorická optimalizace: problém batohu

Zadání

- Na sadách instancí (NK, ZKC, ZKW) experimentálně vyhodnoťte závislost výpočetního času a u všech heuristických algoritmů také relativní chyby (průměrné i maximální) na velikosti instance následujících algoritmů pro
  - konstruktivní verzi problému batohu:
  - Metoda větví a hranic.
  - Metoda dynamického programování (dekompozice podle kapacity nebo podle cen),
  - Jednoduchá greedy heuristika
  - Modifikace této heuristiky (redux), která uvažuje také řešení se sólo nejdražší věcí
  - FPTAS algoritmem, tj. s použitím modifikovaného dynamického programování s dekompozicí podle ceny (při použití dekompozice podle kapacity není algoritmus FPTAS)
- Experiment má odpovědět na tyto otázky:
  - Odpovídají obě závislosti (kvality a času) předpokladům?
  - Je některá heuristická metoda systematicky lepší (tzv. dominance) v některém kritériu?
  - Jak se liší obtížnost jednotlivých sad z hlediska jednotlivých metod?
  - Jaká je závislost maximální chyby (ε) a času FPTAS algoritmu na zvolené přesnosti? Odpovídá předpokladům?

Pokyny

Algoritmy naprogramujte, využijte části programů z minulé úlohy.

Metodu větví a hranic použijte tak, aby omezujícím faktorem byla hodnota optimalizačního kritéria. Tj. použijte ořezávání shora (překročení kapacity batohu) i zdola (stávající řešení nemůže být lepší než nejlepší dosud nalezené).

Pozor! Pokud implementujete FPTAS pomocí zanedbávání bitů, musíte pro daný počet zanedbaných bitů vypočítat max. chybu (ε). V experimentálních výsledcích by počet zanedbaných bitů neměl figurovat, neb neříká nic konkrétního o přesnosti. Pozor, tato max. chyba je jiná pro každou instanci, nezávisí pouze na velikosti instance, ale také na max. ceně.

Pozor! V některých instancích se objevují věci, které svojí hmotností překračují kapacitu batohu. Samozřejmě se jedná o platné instance. Nicméně tyto věci komplikují přepočet cen u FPTAS. Zvažte sami, jak se s tím vypořádat. Řešení je snadné.

Pozn.: u této úlohy je opravdu lepší měřit skutečný CPU čas, namísto počtu konfigurací, jak tomu bylo u předchozího úkolu. Srovnávají se zde principiálně velice odlišné algoritmy, najít jiný relevantní způsob měření složitosti by bylo obtížné (ne-li nemožné).

Zpráva bude obsahovat

- ☒ Popis implementovaných metod.
- ☒ Srovnání výpočetních časů metody větví a hranic, dynamického programování a heuristiky cena/váha (stačí jedna). Grafy vítány.
  - Tj. závislosti výpočetních časů na velikosti instance
- ☒ Porovnání relativních chyb (průměrných a maximálních) obou heuristik.
  - Tj. závislosti rel. chyby na velikosti instance
- ☒ U FPTAS algoritmu pozorujte (naměřte, zdokumentujte) závislost chyby a výpočetního času algoritmu na zvolené přesnosti zobrazení (pro několik různých přesností), srovnání maximální naměřené chyby s teoreticky předpokládanou.
  - Tj. zvolte několik požadovaných přesností (ε), v závislosti na ε měřte čas běhu a reálnou (maximální, případně i průměrnou) chybu algoritmu
- ☒ Zhodnocení naměřených výsledků.

Bonusový bod

Na bonusový bod musí práce obsahovat přínos navíc. Takové přínosy jsou například:

- Srovnání různých dekompozic v dynamickém programování (podle váhy, podle kapacity)
- detailní experimentální analýza FPTAS algoritmu,
- atd.

Řešení

Úkoly předmětu NI-KOP jsem se rozhodl implementovat v jazyce Rust za pomoci nástrojů na *literate programming* – přístup k psaní zdrojového kódu, který upřednostňuje lidsky čitelný popis před seznamem příkazů pro počítač. Tento dokument obsahuje veškerý zdrojový kód nutný k reprodukci mojí práce. Výsledek je dostupný online jako statická webová stránka a ke stažení v PDF.

Instrukce k sestavení programu

Program využívá standardních nástrojů jazyka Rust. O sestavení stačí požádat `cargo`.

```
cd solver
cargo build --release --color always
```

Benchmarking

Od minulého úkolu jsem kompletně přepsal funkcionalitu pro benchmarking, která nově spoléhá na micro benchmark knihovnu `Criterion.rs`. Díky ní stačí pro měření výkonu spustit `cargo bench`. Konkrétní implementace měření je k nahlédnutí v dodatku.

```
uname -a
./cpufetch --logo-short --color ibm
mkdir -p docs/measurements/
cd solver
cargo bench --color always
cp -r target/criterion ../docs/criterion
```

Výsledná měření najdeme ve složce `solver/target/criterion/`. Zahrnují jak hotové reporty jednotlivých algoritmů se srovnáním doby běhu přes různé hodnoty n, tak i detailní záznamy naměřených dat ve formátu JSON.

Srovnání algoritmů

Pro zjednodušení zápisu jsou algoritmy pojmenované zkráceně. Zkratka v přehledu níže odkazuje na podsekci popisující příslušnou implementaci.

- `bb` – metoda větví a hranic
- `dpc` – dynamické programování s rozkladem podle ceny
- `dpw` – dynamické programování s rozkladem podle váhy
- `fptas1` – FPTAS (postavený na `dpc`) pro $\varepsilon = 0.1$
- `fptas2` – FPTAS (postavený na `dpc`) pro $\varepsilon = 0.01$
- `greedy` – hladový algoritmus podle heuristiky poměru cena/váha
- `redux` – `greedy` + řešení pouze s nejdražším předmětem

Zpracování měřených dat pro srovnávací grafy jsem provedl v Pythonu.

```
<<preprocessing>>

<<performance-chart>>
```

```
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import json
import os
from pandas.core.tools.numeric import to_numeric

# load the mean runtime per algorithm. The data is stored in the
# algorithm/n/estimates.json file, where n is the size of the input.
# The mean is in the estimate.mean.point_estimate field.

# TODO: keep this in a better place (duplicities between here and bench.rs)
sets = [ "NK"
       , "ZKC"
       , "ZKW"
       ]

algs = [ "bb"
       , "dpc"
       , "dpw"
       , "fptas1"
       , "fptas2"
       , "greedy"
       , "redux"
       ]

n_values = [4, 10, 15, 20, 22, 25, 27, 30, 32]
data = {}

for s in sets:
    data[s] = {}
    for alg in algs:
        data[s][alg] = {}
        for n in n_values:
            est_file = os.path.join(
```

```
                "solver", "target", "criterion",
                s + "-" + alg, str(n), "new", "estimates.json"
            )
            if os.path.exists(est_file):
                with open(est_file, "r") as f:
                    estimates = json.load(f)
                    mean = estimates["mean"]["point_estimate"]
                    data[s][alg][n] = { "mean": mean / 1000 / 1000 / 1000
                                      }
                err_file = os.path.join(
                    "docs", "measurements",
                    s + "_" + alg + "_" + str(n) + ".txt"
                )
                with open(err_file, "r") as f:
                    measurements = pd.read_csv(f)
                    data[s][alg][n]["error"] = { "max": measurements["max"]
                                               , "avg": measurements["avg"]
                                               }

# plot the mean runtimes and max errors

figsize = (14, 8)
for s in sets:
    fig, ax = plt.subplots(figsize = figsize)
    plt.title("Průměrná doba běhu")
    plt.xlabel("Velikost instance")
    plt.ylabel("Průměrná doba běhu (sec)")
    plt.xticks(n_values)
    for alg in algs:
        plt.plot([n for n in data[s][alg]], [data[s][alg][n]["mean"] for n in data[s][alg]], "--o", l
    plt.legend()
    plt.savefig("docs/assets/{}_mean_runtimes.svg".format(s))

    fig, ax = plt.subplots(figsize = figsize)
    plt.title("Závislost maximální chyby na velikosti instance")
    plt.xlabel("Velikost instance")
    plt.ylabel("Maximální chyba")
    plt.xticks(n_values)
    yticks = np.append(ax.get_yticks(), [0.1, 0.01])
    ax.set_yticks(yticks)
    ax.grid(linestyle = "dotted")
    for alg in algs:
        plt.plot([n for n in data[s][alg]], [data[s][alg][n]["error"]["max"] for n in data[s][alg]],
    plt.legend()
    plt.savefig("docs/assets/{}_max_errors.svg".format(s))
```

Výkon každého algoritmu byl změřen na stovce různých zadání z jedné datové sady alespoň desetkrát za sebou (přesný počet spuštění řídí knihovna v závislosti na výkonu algoritmu). Vyobrazený čas je proto stokrát vyšší, než skutečná doba řešení jedné instance problému dané velikosti.

Graf maximální chyby zvýrazňuje hranice požadované od FPTAS algoritmu. Zajímavé je, že maximální chyba je relativně hluboko pod požadovanou horní mezí.

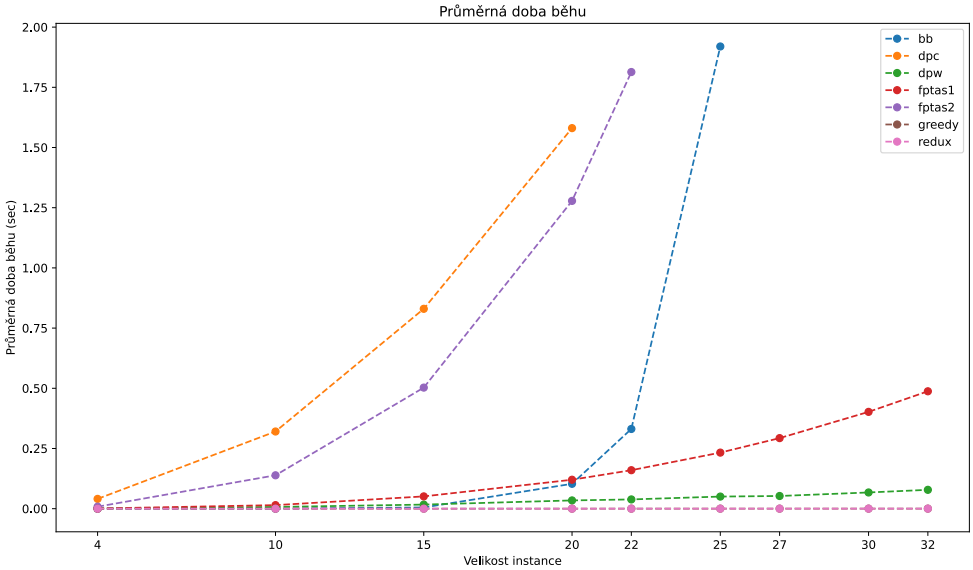

Figure 1: NK: Závislost doby běhu na počtu předmětů.

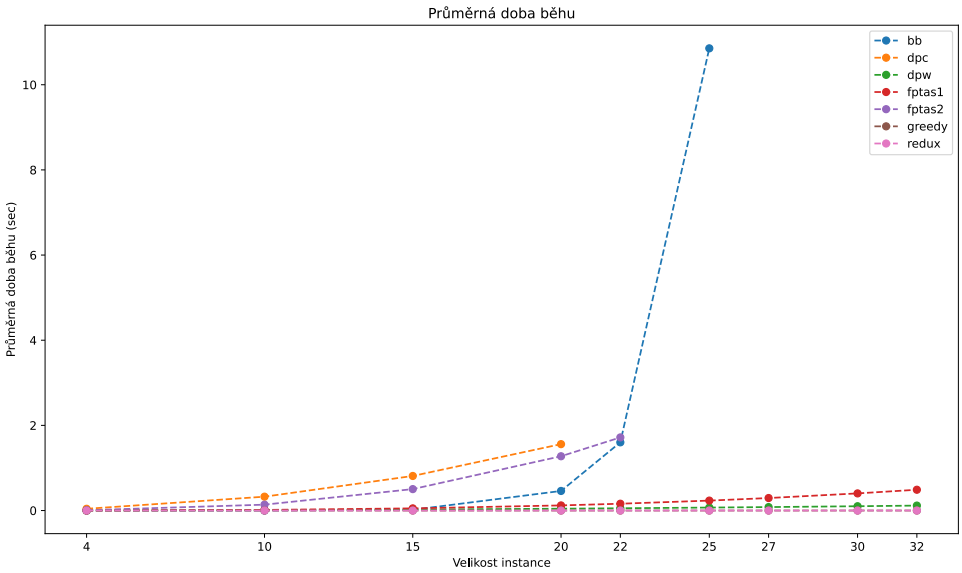

Figure 2: ZKC: Závislost doby běhu na počtu předmětů.

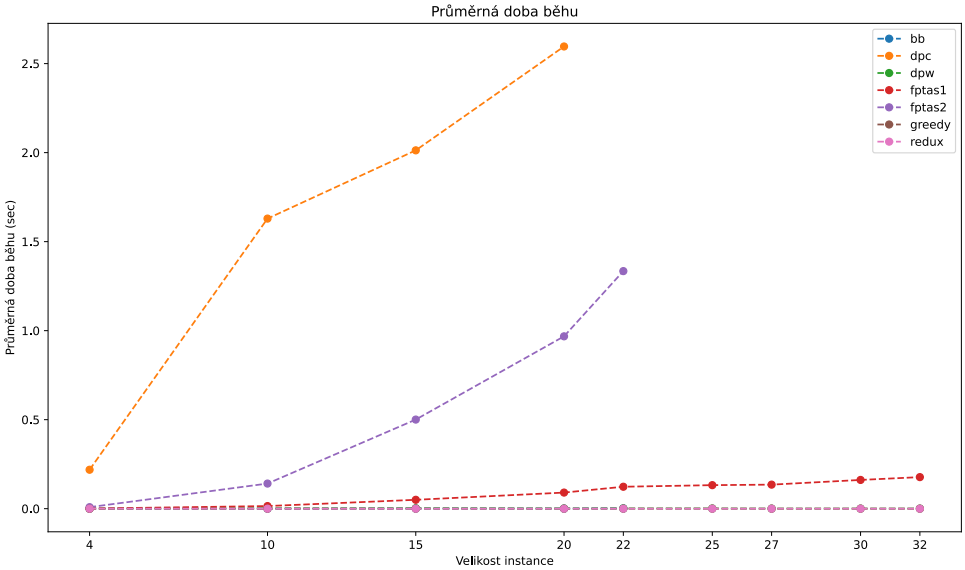

Figure 3: ZKW: Závislost doby běhu na počtu předmětů.

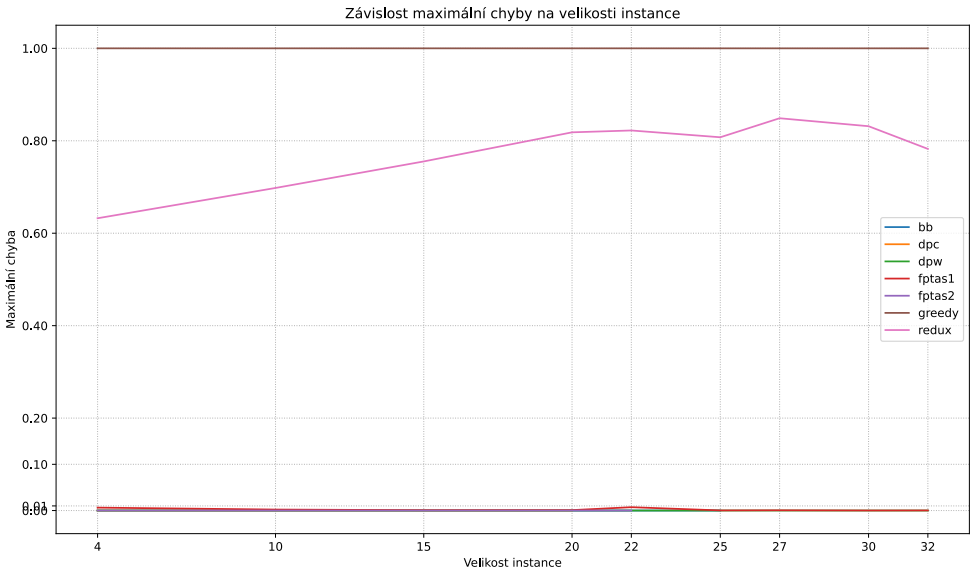

Figure 4: NK: Maximální chyba řešení.

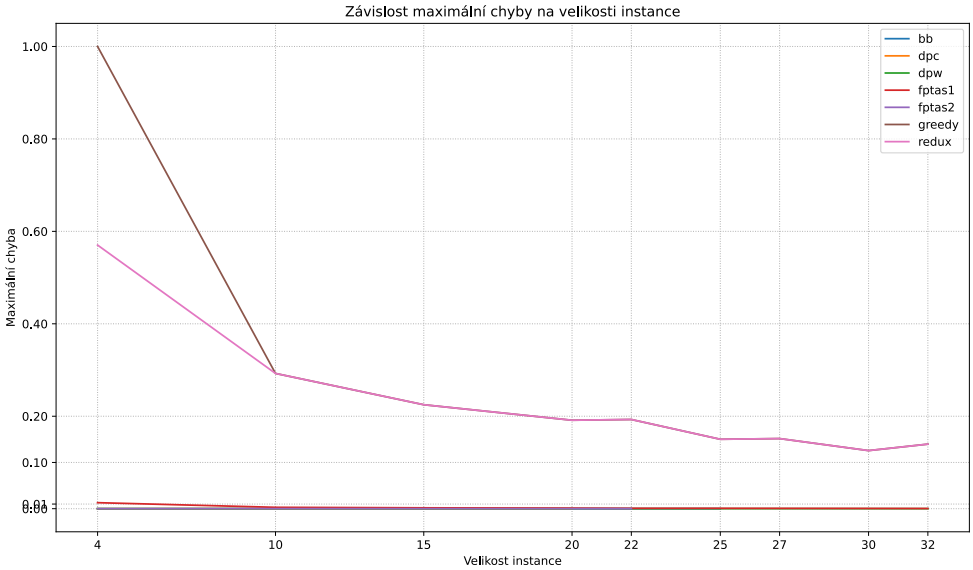

Figure 5: ZKC: Maximální chyba řešení.

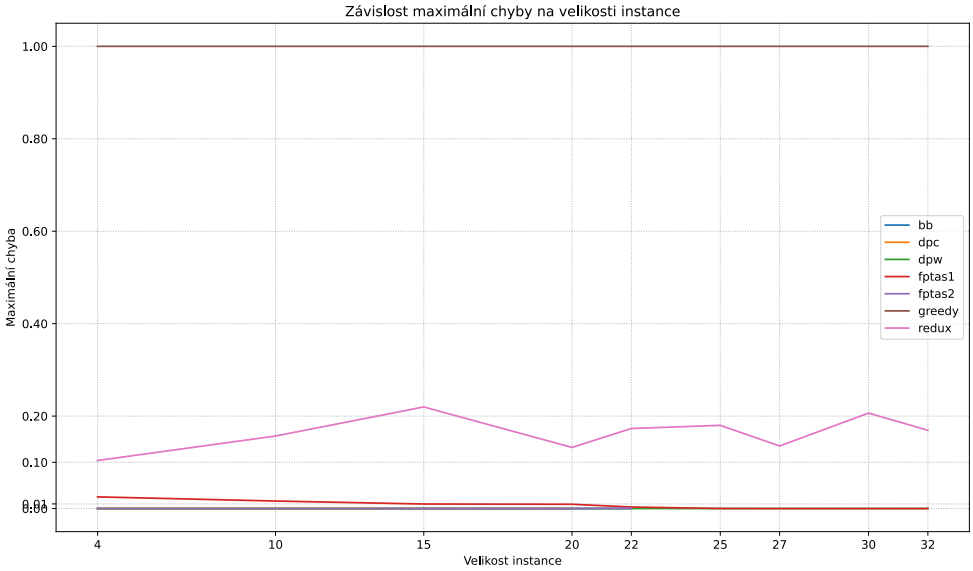

Figure 6: ZKW: Maximální chyba řešení.

Analýza

Detailní analýza algoritmu FPTAS je rozdělená podle sady instancí a parametru ε.

- sada NK
  - $\varepsilon = 0.1$ `NK-fptas1`
  - $\varepsilon = 0.01$ `NK-fptas2`
- sada ZKC
  - $\varepsilon = 0.1$ `ZKC-fptas1`
  - $\varepsilon = 0.01$ `ZKC-fptas2`
- sada ZKW
  - $\varepsilon = 0.1$ `ZKW-fptas1`
  - $\varepsilon = 0.01$ `ZKW-fptas2`

Z grafů je vidět, že (alespoň na těchto datových sadách) se dekompozice podle váhy vyplatí mnohem více, než dekompozice podle ceny. FPTAS sice něco z výpočetní náročnosti ušetří (na úkor kvality řešení), dynamické programování podle váhy ale stále vede.

Odpovídají obě závislosti (kvality a času) předpokladům? Ne. Čekal jsem, že FPTAS bude často mnohem blíže požadované hranici ε. Varianta `fptas1` ale většinou překoná i hranici pro `fptas2`, přitom je mnohem rychlejší.

V sadě ZKW je zároveň poměrně málo instancí, ve kterých se navíc objevují nečekané trendy. Ty vedou k tomu, že dynamické programování s rozkladem podle ceny na této sadě projevuje nemonotonní vztah mezi velikostí instance a dobou běhu.

Je některá heuristická metoda systematicky lepší v některém kritériu? Heuristiky běží podstatně rychleji než ostatní algoritmy. Redux je navíc i užitečná, protože její maximální chyba je shora omezená. Naproti tomu hloupý hladový přístup často skončí s vysokou chybou.

Jak se liší obtížnost jednotlivých sad z hlediska jednotlivých metod? Sady NK a ZKC jsou si zřejmě dost podobné. Metoda větví a hranic má trochu problémy v sadě ZKC, pro $n = 25$ už řešení trvalo 10 sekund (proto jsem jej z měření vyřadil). Sada ZKW se zdá být podstatně jednodušší pro dynamické programování a FPTAS na něm založený.

Jaká je závislost maximální chyby (ε) a času FPTAS algoritmu na zvolené přesnosti? Odpovídá předpokladům? Nikoliv. `fptas1` (pro $\varepsilon = 0.1$) je překvapivě časově efektivní a zároveň dosahuje velmi pěkných výsledků. Kromě sady ZKW, kde má trochu problémy, často přesáhne mez stanovenou jeho výrazně pomalejšímu bratru `fptas2`. Na těchto datech se zdá být dobrým kompromisem mezi výpočetní složitostí a přesností výsledku, ačkoliv kvality dynamického programování podle váhy samozřejmě nedosahuje.

Implementace

Program začíná definicí datové struktury reprezentující instanci problému batohu.

```
#[derive(Debug, PartialEq, Eq, Clone)]
pub struct Instance {
    pub id: i32, m: u32, pub items: Vec<(u32, u32)>
}
```

Následující úryvek poskytuje ptačí pohled na strukturu souboru. Použité knihovny jsou importovány na začátku, následuje již zmíněná definice instance problému, dále funkce `main()`, parser, definice struktury řešení a její podpůrné funkce, samotné algoritmy řešiče a v neposlední řadě sada automatických testů.

```
<<imports>>

<<algorithm-map>>

pub fn solve_stream<T>(
    alg: for <'b> fn(&'b Instance) -> Solution<'b>,
    solutions: HashMap<(u32, i32), OptimalSolution>,
    stream: &mut T
) -> Result<Vec<(u32, f64)>> where T: BufRead {
    let mut results = vec![];
    loop {
        match parse_line(stream)?.as_ref().map(|inst| (inst, alg(inst))) {
            Some((inst, sln)) => {
                let optimal = &solutions[&(inst.items.len() as u32, inst.id)];
                let error = 1.0 - sln.cost as f64 / optimal.cost as f64;
                results.push((sln.cost, error))
            },
            None => return Ok(results)
        }
    }
}

use std::result::Result as IOResult;
pub fn list_input_files(set: &str, r: Range<u32>) -> Result<Vec<IOResult<DirEntry, std::io::Error>>>
    let f = |res: &IOResult<DirEntry, std::io::Error> | res.as_ref().ok().filter(|f| {
        let file_name = f.file_name();
        let file_name = file_name.to_str().unwrap();
        // keep only regular files
        f.file_type().unwrap().is_file() &&
        // ... whose names start with the set name,
        file_name.starts_with(set) &&
        // ... continue with an integer between 0 and 15,
        file_name[set.len()..]
        .split('_').next().unwrap().parse::<u32>().ok()
        .filter(|n| r.contains(n)).is_some() &&
        // ... and end with `_inst.dat` (for "instance").
        file_name.ends_with("_inst.dat")
    }).is_some();
    Ok(read_dir("./ds/")?.filter(f).collect())
}

<<problem-instance-definition>>

<<solution-definition>>

<<parser>>

impl Instance {
    <<solver-dpw>>

    <<solver-dpc>>
```

```
    <<solver-fptas>>

    <<solver-greedy>>

    <<solver-greedy-redux>>

    <<solver-bb>>

    <<solver-bf>>
}

<<tests>>
```

Řešení v podobě datové struktury `Solution` má kromě reference na instanci problému především bit array udávající množinu předmětů v pomyslném batohu. Zároveň nese informaci o počtu navštívených konfigurací při jeho výpočtu.

```
pub type Config = BitArr!(for 64);

#[derive(PartialEq, Eq, Clone, Copy, Debug)]
pub struct Solution<'a> { weight: u32, pub cost: u32, cfg: Config, pub inst: &'a Instance }

#[derive(Debug, PartialEq, Eq, Clone)]
pub struct OptimalSolution { id: i32, pub cost: u32, cfg: Config }

<<solution-helpers>>
```

Protože se strukturami typu `Solution` se v algoritmech pracuje hojně, implementoval jsem pro ně koncept řazení a pomocné metody k počítání navštívených konfigurací a přidávání předmětů do batohu.

```
impl <'a> PartialOrd for Solution<'a> {
    fn partial_cmp(&self, other: &Self) -> Option<cmp::Ordering> {
        use cmp::Ordering;
        let Solution {weight, cost, ..} = self;
        Some(match cost.cmp(&other.cost) {
            Ordering::Equal => weight.cmp(&other.weight).reverse(),
            other => other,
        })
    }
}

impl <'a> Ord for Solution<'a> {
    fn cmp(&self, other: &Self) -> cmp::Ordering {
        self.partial_cmp(other).unwrap()
    }
}

impl <'a> Solution<'a> {
    fn with(mut self, i: usize) -> Solution<'a> {
        let (w, c) = self.inst.items[i];
        if !self.cfg[i] {
            self.cfg.set(i, true);
            self.weight += w;
            self.cost += c;
```

```
        }
        self
    }

    fn default(inst: &'a Instance) -> Solution<'a> {
        Solution { weight: 0, cost: 0, cfg: Config::default(), inst }
    }

    fn overweight(inst: &'a Instance) -> Solution<'a> {
        Solution { weight: u32::MAX, cost: 0, cfg: Config::default(), inst }
    }
}
```

Algoritmy

Aby bylo k jednotlivým implementacím jednoduché přistupovat, všechny implementované algoritmy jsou uloženy pod svými názvy v `BTreeMap`ě. Tu používáme při vybírání algoritmu pomocí argumentu předaného na příkazové řádce, v testovacím kódu na testy všech implementací atp.

```
pub fn get_algorithms() -> BTreeMap<&'static str, fn(&Instance) -> Solution> {
    let cast = |x: fn(&Instance) -> Solution| x;
    // the BTreeMap works as a trie, maintaining alphabetic order
    BTreeMap::from([
        ("bf",     cast(Instance::brute_force)),
        ("bb",     cast(Instance::branch_and_bound)),
        ("dpc",    cast(Instance::dynamic_programming_c)),
        ("dpw",    cast(Instance::dynamic_programming_w)),
        ("fptas1", cast(|inst| inst.fptas(10f64.powi(-1)))),
        ("fptas2", cast(|inst| inst.fptas(10f64.powi(-2)))),
        ("greedy", cast(Instance::greedy)),
        ("redux",  cast(Instance::greedy_redux)),
    ])
}
```

Hladový přístup Implementace hladové strategie využívá knihovny `permutation`. Problém ve skutečnosti řešíme na isomorfní instanci, která má předměty uspořádané. Jediné, co se změní, je pořadí, ve kterém předměty navštěvujeme. Proto stačí aplikovat řadicí permutaci předmětů na posloupnost indexů, které procházíme. Přesně to dělá výraz `(0..items.len()).map(ord)`.

```
fn greedy(&self) -> Solution {
    use ::permutation::*;
    let Instance {m, items, ..} = self;
    fn ratio((w, c): (u32, u32)) -> f64 {
        let r = c as f64 / w as f64;
        if r.is_nan() { f64::NEG_INFINITY } else { r }
    }
    let permutation = sort_by(
        &(items)[..],
        |a, b|
            ratio(*a)
            .partial_cmp(&ratio(*b))
            .unwrap()
            .reverse() // max item first
```

```
    );
    let ord = { #[inline] |i| permutation.apply_idx(i) };

    let mut sol = Solution::default(self);
    for i in (0..items.len()).map(ord) {
        let (w, _c) = items[i];
        if sol.weight + w <= *m {
            sol = sol.with(i);
        } else { break }
    }

    sol
}
```

Hladový přístup – redux Redux verze hladové strategie je více méně deklarativní. Výsledek redux algoritmu je maximum z hladového řešení a řešení sestávajícího pouze z nejdražšího předmětu. K indexu nejdražšího předmětu dojdeme tak, že sepneme posloupnosti indexů a předmětů, vyřadíme prvky, jejichž váha přesahuje kapacitu batohu a vybereme maximální prvek podle ceny.

```
fn greedy_redux(&self) -> Solution {
    let greedy = self.greedy();
    (0_usize..)
        .zip(self.items.iter())
        .filter(|(_, (w, _))| *w <= self.m)
        .max_by_key(|(_, (_, c))| c)
        .map(|(highest_price_index, _)|
            max(greedy, Solution::default(self).with(highest_price_index))
        ).unwrap_or(greedy)
}
```

Hrubá síla

```
fn brute_force(&self) -> Solution {
    fn go<'a>(items: &'a [(u32, u32)], current: Solution<'a>, i: usize, m: u32) -> Solution<'a> {
        if i >= items.len() { return current }

        let (w, _c) = items[i];
        let next = |current, m| go(items, current, i + 1, m);
        let include = || {
            let current = current.with(i);
            next(current, m - w)
        };
        let exclude = || next(current, m);

        if w <= m {
            max(include(), exclude())
        }
        else { exclude() }
    }

    go(&self.items, Solution::default(self), 0, self.m)
}
```

Branch & bound

```rust
fn branch_and_bound(&self) -> Solution {
    struct State<'a>(&'a Vec<(u32, u32)>, Vec<u32>);
    let prices: Vec<u32> = {
        self.items.iter().rev()
        .scan(0, |sum, (_w, c)| {
            *sum += c;
            Some(*sum)
        })
        .collect::<Vec<_>>().into_iter().rev().collect()
    };

    fn go<'a>(state: &'a State, current: Solution<'a>, best: Solution<'a>, i: usize, m: u32) -> Solut
        let State(items, prices) = state;
        if i >= items.len() || current.cost + prices[i] <= best.cost {
            return current
        }

        let (w, _c) = items[i];
        let next = |current, best, m| go(state, current, best, i + 1, m);
        let include = || {
            let current = current.with(i);
            next(current, max(current, best), m - w)
        };
        let exclude = |best: Solution<'a>| next(current, best, m);

        if w <= m {
            let x = include();
            max(x, exclude(x))
        }
        else { exclude(best) }
    }

    // FIXME borrowck issues
    let state = State(&self.items, prices);
    let empty = Solution::default(self);
    Solution { inst: self, ..go(&state, empty, empty, 0, self.m) }
}
```

Dynamické programování Jako první jsem naimplementoval dynamické programování s rozkladem podle váhy, jehož časová složitost je $\Theta(nM)$, kde M je kapacita batohu. Popsal jsem ho už v minulém úkolu.

```rust
fn dynamic_programming_w(&self) -> Solution {
    let Instance {m, items, ..} = self;
    let mut next = vec![Solution::default(self); *m as usize + 1];
    let mut last = vec![];

    for (i, &(weight, _cost)) in items.iter().enumerate() {
        last.clone_from(&next);

        for cap in 0 ..= *m as usize {
```

```
            let s = if (cap as u32) < weight {
                    last[cap]
                } else {
                    let rem_weight = max(0, cap as isize - weight as isize) as usize;
                    max(last[cap], last[rem_weight].with(i))
                };
            next[cap] = s;
        }
    }

    *next.last().unwrap()
}
```

Následně jsem (pro FPTAS) implementoval dynamické programování s rozkladem podle ceny, které je adaptací algoritmu výše. Narozdíl od předchozího algoritmu je tady výchozí hodnotou v tabulce efektivně nekonečná váha, kterou se snažíme minimalizovat. K reprezentaci řešení s nekonečnou vahou používám přidruženou funkci `Solution::overweight`, která vrátí neplatné řešení s váhou $2^{32} - 1$. Pokud na něj v průběhu výpočtu algoritmus narazí, předá jej dál jako `Solution::default` (vždy v nejlevějším sloupci DP tabulky, tedy `last[0]`), aby při přičtení váhy uvažovaného předmětu nedošlo k přetečení.

O výběr řešení minimální váhy se stará funkce `max`, neboť implementace uspořádání pro typ `Solution` řadí nejprve vzestupně podle ceny a následně sestupně podle váhy. V tomto případě porovnáváme vždy dvě řešení stejných cen (a nebo je `last[cap]` neplatné řešení s nadváhou, které má cenu 0).

```
fn dynamic_programming_c(&self) -> Solution {
    let Instance {items, ..} = self;
    let max_profit = items.iter().map(|(_, c)| *c).max().unwrap() as usize;
    let mut next = vec![Solution::overweight(self); max_profit * items.len() + 1];
    let mut last = vec![];
    next[0] = Solution::default(self);

    for (i, &(_weight, cost)) in items.iter().enumerate() {
        last.clone_from(&next);

        for cap in 1 ..= max_profit * items.len() {
            let s = if (cap as u32) < cost {
                    last[cap]
                } else {
                    let rem_cost = (cap as isize - cost as isize) as usize;
                    let lightest_for_cost = if last[rem_cost].weight == u32::MAX {
                        last[0] // replace the overweight solution with the empty one
                    } else { last[rem_cost] };

                    max(last[cap], lightest_for_cost.with(i))
                };
            next[cap] = s;
        }
    }

    *next.iter().filter(|sln| sln.weight <= self.m).last().unwrap()
}
```

FPTAS FPTAS algoritmus přeškáluje ceny předmětů a následně spustí dynamické programování s rozkladem podle ceny na upravenou instanci problému. V řešení stačí opravit referenci výchozí instance (`inst: self`) a přepočíst cenu podle vypočítané konfigurace, samotné indexy předmětů se škálováním nemění.

```
// TODO: are items heavier than the knapsack capacity a problem? if so, we
// can just zero them out
fn fptas(&self, eps: f64) -> Solution {
    let Instance {m: _, items, ..} = self;
    let max_profit = items.iter().map(|(_, c)| *c).max().unwrap();
    let scaling_factor = eps * max_profit as f64 / items.len() as f64;
    let items: Vec<(u32, u32)> = items.iter().map(|(w, c)|
        (*w, (*c as f64 / scaling_factor).floor() as u32
    )).collect();

    let iso = Instance { items, ..*self };
    let sln = iso.dynamic_programming_c();
    let cost = (0usize..).zip(self.items.iter()).fold(0, |acc, (i, (_w, c))|
        acc + sln.cfg[i] as u32 * c
    );
    Solution { inst: self, cost, ..sln }
}
```

Závěr

Zadání jsem se znovu pokusil splnit v celém rozsahu. Implementoval jsem všechny zadané algoritmy, k nim navíc dynamické programování s rozkladem podle kapacity. Díky nové implementaci měření jsem zachytil podrobné statistiky ze všech experimentů, včetně středních odchylek, průměrů, mediánů a dalších statistických veličin doby běhu jednotlivých algoritmů na různých datových sadách, ale také průměrné a maximální chyby všech výsledků.

Podrobné reporty kombinace algoritmů, datových sad a velikostí instancí, jakožto i přehledy výkonu algoritmů na datových sadách napříč všemi velikostmi instancí jsou zachycené zvlášť. Archív odevzdaný na Moodle je obsahuje v podsložce `criterion`, jsou také dostupné online (např. přehled výkonu `redux` na ZKC).

Appendix

Dodatek obsahuje nezajímavé části implementace, jako je import symbolů z knihoven.

```
use std::{cmp, cmp::max,
    ops::Range,
    str::FromStr,
    io::{BufRead, BufReader},
    collections::{BTreeMap, HashMap},
    fs::{read_dir, File, DirEntry},
};
use anyhow::{Context, Result, anyhow};
use bitvec::prelude::BitArr;

#[cfg(test)]
#[macro_use(quickcheck)]
extern crate quickcheck_macros;
```

Zpracování vstupu zajišťuje jednoduchý parser pracující řádek po řádku. Pro testy je tu parser formátu souborů s optimálními řešeními.

```
<<boilerplate>>

pub fn parse_line<T>(stream: &mut T) -> Result<Option<Instance>> where T: BufRead {
    let mut input = String::new();
    if stream.read_line(&mut input)? == 0 {
        return Ok(None)
    }

    let mut  numbers = input.split_whitespace();
    let id = numbers.parse_next()?;
    let  n = numbers.parse_next()?;
    let  m = numbers.parse_next()?;

    let mut items: Vec<(u32, u32)> = Vec::with_capacity(n);
    for _ in 0..n {
        let w = numbers.parse_next()?;
        let c = numbers.parse_next()?;
        items.push((w, c));
    }

    Ok(Some(Instance {id, m, items}))
}

fn parse_solution_line<T>(mut stream: T) -> Result<Option<OptimalSolution>> where T: BufRead {
    let mut input = String::new();
    if stream.read_line(&mut input)? == 0 {
        return Ok(None)
    }

    let mut    numbers = input.split_whitespace();
    let   id = numbers.parse_next()?;
    let    n = numbers.parse_next()?;
    let cost = numbers.parse_next()?;

    let mut items = Config::default();
    for i in 0..n {
        let a: u8 = numbers.parse_next()?;
        items.set(i, a == 1);
    }

    Ok(Some(OptimalSolution {id, cost, cfg: items}))
}

pub fn load_solutions(set: &str) -> Result<HashMap<(u32, i32), OptimalSolution>> {
    let mut solutions = HashMap::new();

    let files = read_dir("../data/constructive/")?
        .filter(|res| res.as_ref().ok().filter(|f| {
            let name = f.file_name().into_string().unwrap();
```

```
            f.file_type().unwrap().is_file() &&
            name.starts_with(set) &&
            name.ends_with("_sol.dat")
        }).is_some());

    for file in files {
        let file = file?;
        let n = file.file_name().into_string().unwrap()[set.len()..].split('_').next().unwrap().parse
        let mut stream = BufReader::new(File::open(file.path())?);
        while let Some(opt) = parse_solution_line(&mut stream)? {
            solutions.insert((n, opt.id), opt);
        }
    }

    Ok(solutions)
}
```

Trait `Boilerplate` definuje funkci `parse_next` pro zkrácení zápisu zpracování vstupu.

```
trait Boilerplate {
    fn parse_next<T: FromStr>(&mut self) -> Result<T>
      where <T as FromStr>::Err: std::error::Error + Send + Sync + 'static;
}

impl Boilerplate for std::str::SplitWhitespace<'_> {
    fn parse_next<T: FromStr>(&mut self) -> Result<T>
      where <T as FromStr>::Err: std::error::Error + Send + Sync + 'static {
        let str = self.next().ok_or_else(|| anyhow!("unexpected end of input"))?;
        str.parse::<T>()
           .with_context(|| format!("cannot parse {}", str))
    }
}
```

Měření výkonu

Benchmark postavený na knihovně `Criterion.rs` se nachází v souboru níže.

```
extern crate solver;

use solver::*;
use anyhow::{Result, anyhow};
use std::{collections::HashMap, fs::File, io::{BufReader, Write}, ops::Range, time::Duration};
use criterion::{criterion_group, criterion_main, Criterion, BenchmarkId};

fn full(c: &mut Criterion) -> Result<()> {
    let algs = get_algorithms();
    let mut solutions = HashMap::new();
    let ranges = HashMap::from([
        ("bb",     0..=25),
        ("dpw",    0..=32),
        ("dpc",    0..=20),
        ("fptas1", 0..=32),
        ("fptas2", 0..=22),
        ("greedy", 0..=32),
```

```
        ("redux",  0..=32),
    ]);

    let mut input: HashMap<(&str, u32), Vec<Instance>> = HashMap::new();
    let ns = [4, 10, 15, 20, 22, 25, 27, 30, 32];
    let sets = ["NK", "ZKC", "ZKW"];
    for set in sets {
        solutions.insert(set, load_solutions(set)?);
        for n in ns {
            input.insert((set, n), load_input(set, n .. n + 1)?
                .into_iter()
                .rev()
                .take(100)
                .collect()
            );
        }
    }

    benchmark(algs, c, &ns, &sets, ranges, solutions, input)?;
    Ok(())
}

fn benchmark(
    algs: std::collections::BTreeMap<&str, fn(&Instance) -> Solution>,
    c: &mut Criterion,
    ns: &[u32],
    sets: &[&'static str],
    ranges: HashMap<&str, std::ops::RangeInclusive<u32>>,
    solutions: HashMap<&str, HashMap<(u32, i32), OptimalSolution>>,
    input: HashMap<(&str, u32), Vec<Instance>>
) -> Result<(), anyhow::Error> {
    Ok(for set in sets {
        for (name, alg) in algs.iter() {
            let mut group = c.benchmark_group(format!("{}-{}", set, name));
            group.sample_size(10).warm_up_time(Duration::from_millis(200));

            for n in ns {
                if !ranges.get(*name).filter(|r| r.contains(&n)).is_some()
                || (*name == "bb" && *n > 22 && *set == "ZKW") {
                    continue;
                }

                let (max, avg, nonzero_n) =
                    measure(&mut group, *alg, &solutions[set], *n, &input[&(*set, *n)]);
                eprintln!("max: {}, avg: {}, n: {} vs real n: {}", max, avg, nonzero_n, n);
                let avg = avg / nonzero_n as f64;

                let mut file = File::create(format!("../docs/measurements/{}_{}_{}.txt", set, name, n
                file.write_all(format!("max,avg\n{},{}", max, avg).as_bytes())?;
            }
            group.finish();
        }
```

```
    })
}

fn measure(
    group: &mut criterion::BenchmarkGroup<criterion::measurement::WallTime>,
    alg: for<'a> fn(&'a Instance) -> Solution<'a>,
    solutions: &HashMap<(u32, i32), OptimalSolution>,
    n: u32,
    instances: &Vec<Instance>
) -> (f64, f64, u32) {
    let mut stats = (0.0, 0.0, 0);
    group.bench_with_input(
        BenchmarkId::from_parameter(n),
        instances,
        |b, ins| b.iter(
            || ins.iter().for_each(|inst| {
                let sln = alg(inst);
                let optimal = &solutions[&(n, inst.id)];
                if optimal.cost != 0 {
                    let error = 1.0 - sln.cost as f64 / optimal.cost as f64;
                    let (max, avg, n) = stats;
                    stats = (if error > max { error } else { max }, avg + error, n + 1);
                }
            })
        )
    );

    stats
}

fn load_input(set: &str, r: Range<u32>) -> Result<Vec<Instance>> {
    let mut instances = Vec::new();

    for file in list_input_files(set, r)? {
        let file = file?;
        let mut r = BufReader::new(File::open(file.path())?);
        while let Some(inst) = parse_line(&mut r)? {
            instances.push(inst);
        }
    }

    Ok(instances)
}

fn proxy(c: &mut Criterion) {
    full(c).unwrap()
}

criterion_group!(benches, proxy);
criterion_main!(benches);
```

Spouštění jednotlivých řešičů

Projekt podporuje sestavení spustitelného souboru schopného zpracovat libovolný vstup ze zadání za pomoci algoritmu zvoleného na příkazové řádce. Zdrojový kód tohoto rozhraní se nachází v souboru `solver/src/bin/main.rs`. Na standardní výstup vypisuje cenu a chybu řešení, spoléhá ovšem na to, že mezi optimálními řešeními najde i to pro kombinaci velikosti a ID zadané instance.

```
extern crate solver;

use std::io::stdin;
use solver::*;
use anyhow::{Result, anyhow};

fn main() -> Result<()> {
    let algorithms = get_algorithms();
    let solutions = load_solutions("NK")?;

    let alg = *{
        <<select-algorithm>>
    }?;

    for (cost, error) in solve_stream(alg, solutions, &mut stdin().lock())? {
        println!("{} {}", cost, error);
    }
    Ok(())
}
```

Funkci příslušnou vybranému algoritmu vrátíme jako hodnotu tohoto bloku:

```
let args: Vec<String> = std::env::args().collect();
if args.len() == 2 {
    let alg = &args[1][..];
    if let Some(f) = algorithms.get(alg) {
        Ok(f)
    } else {
        Err(anyhow!("\"{}\" is not a known algorithm", alg))
    }
} else {
    println!(
        "Usage: {} <algorithm>\n\twhere <algorithm> is one of {}",
        args[0],
        algorithms.keys().map(ToString::to_string).collect::<Vec<_>>().join(", ")
    );
    Err(anyhow!("Expected 1 argument, got {}", args.len() - 1))
}
```

Automatické testy

Implementaci doplňují automatické testy k ověření správnosti, včetně property-based testu s knihovnou quickcheck.

```
#[cfg(test)]
mod tests {
    use super::*;
    use quickcheck::{Arbitrary, Gen};
```

```rust
use std::{fs::File, io::BufReader};

impl Arbitrary for Instance {
    fn arbitrary(g: &mut Gen) -> Instance {
        Instance {
            id:    i32::arbitrary(g),
            m:     u32::arbitrary(g).min(10_000),
            items: vec![<(u32, u32)>::arbitrary(g)]
                       .into_iter()
                       .chain(Vec::arbitrary(g).into_iter())
                       .take(10)
                       .map(|(w, c): (u32, u32)| (w.min(10_000), c % 10_000))
                       .collect(),
        }
    }

    fn shrink(&self) -> Box<dyn Iterator<Item = Self>> {
        let data = self.clone();
        let chain: Vec<Instance> = quickcheck::empty_shrinker()
            .chain(self.id   .shrink().map(|id   | Instance {id,    ..(&data).clone()}))
            .chain(self.m    .shrink().map(|m    | Instance {m,     ..(&data).clone()}))
            .chain(self.items.shrink().map(|items| Instance { items, ..(&data).clone() })
                    .filter(|i| !i.items.is_empty()))
            .collect();
        Box::new(chain.into_iter())
    }
}

impl <'a> Solution<'a> {
    fn assert_valid(&self) {
        let Solution { weight, cost, cfg, inst } = self;
        let Instance { m, items, .. } = inst;

        let (computed_weight, computed_cost) = items
            .into_iter()
            .zip(cfg)
            .map(|((w, c), b)| {
                if *b { (*w, *c) } else { (0, 0) }
            })
            .reduce(|(a0, b0), (a1, b1)| (a0 + a1, b0 + b1))
            .unwrap_or_default();

        assert!(computed_weight <= *m);
        assert_eq!(computed_cost, *cost);
        assert_eq!(computed_weight, *weight);
    }
}

#[test]
fn stupid() {
    // let i = Instance { id: 0, m: 1, b: 0, items: vec![(1, 0), (1, 0)] };
    // i.branch_and_bound2().assert_valid(&i);
```

```rust
    let i = Instance { id: 0, m: 1, items: vec![(1, 1), (1, 2), (0, 1)] };
    let bb = i.branch_and_bound();
    assert_eq!(bb.cost, i.dynamic_programming_w().cost);
    assert_eq!(bb.cost, i.dynamic_programming_c().cost);
    assert_eq!(bb.cost, i.greedy_redux().cost);
    assert_eq!(bb.cost, i.brute_force().cost);
    assert_eq!(bb.cost, i.greedy().cost);
}

#[test]
fn proper() -> Result<()> {
    type Solver = (&'static str, for<'a> fn(&'a Instance) -> Solution<'a>);
    let algs = get_algorithms();
    let algs: Vec<Solver> = algs.iter().map(|(s, f)| (*s, *f)).collect();
    let opts = load_solutions("NK")?;
    println!("loaded {} optimal solutions", opts.len());

    let solve: for<'a> fn(&Vec<_>, &'a _) -> Vec<(&'static str, Solution<'a>)> =
        |algs, inst|
        algs.iter().map(|(name, alg): &Solver| (*name, alg(inst))).collect();

    let mut files = list_input_files("NK", 0..5)?.into_iter();
    // make sure `files` is not empty
    let first = files.next().ok_or(anyhow!("no instance files loaded"))?;
    for file in vec![first].into_iter().chain(files) {
        let file = file?;
        println!("Testing {}", file.file_name().to_str().unwrap());
        // open the file
        let mut r = BufReader::new(File::open(file.path())?);
        // solve each instance with all algorithms
        while let Some(slns) = parse_line(&mut r)?.as_ref().map(|x| solve(&algs, x)) {
            // verify correctness
            slns.iter().for_each(|(alg, s)| {
                eprint!("\rid: {} alg: {}\t", s.inst.id, alg);
                s.assert_valid();
                let key = (s.inst.items.len() as u32, s.inst.id);
                assert!(s.cost <= opts[&key].cost);
            });
        }
    }
    Ok(())
}

#[test]
fn dpc_simple() {
    let i = Instance { id: 0, m: 0, items: vec![(0, 1), (0, 1)] };
    let s = i.dynamic_programming_c();
    assert_eq!(s.cost, 2);
    assert_eq!(s.weight, 0);
    s.assert_valid();
}
```

```rust
#[test]
fn fptas_is_within_bounds() -> Result<()> {
    let opts = load_solutions("NK")?;
    for eps in [0.1, 0.01] {
        for file in list_input_files("NK", 0..5)? {
            let file = file?;
            let mut r = BufReader::new(File::open(file.path())?);
            while let Some(sln) = parse_line(&mut r)?.as_ref().map(|x| x.fptas(eps)) {
                // make sure the solution from fptas is at least (1 - eps) * optimal cost
                let key = (sln.inst.items.len() as u32, sln.inst.id);
                println!("{} {} {}", sln.cost, opts[&key].cost, (1.0 - eps) * opts[&key].cost as
                assert!(sln.cost as f64 >= opts[&key].cost as f64 * (1.0 - eps));
            }
        }
    }
    Ok(())
}

#[test]
fn small_bb_is_correct() {
    let a = Instance {
        id: -10,
        m: 165,
        items: vec![ (86,  744)
                   , (214, 1373)
                   , (236, 1571)
                   , (239, 2388)
                   ],
    };
    a.branch_and_bound().assert_valid();
}

#[test]
fn bb_is_correct() -> Result<()> {
    use std::fs::File;
    use std::io::BufReader;
    let inst = parse_line(
        &mut BufReader::new(File::open("ds/NK15_inst.dat")?)
    )?.unwrap();
    println!("testing {:?}", inst);
    inst.branch_and_bound().assert_valid();
    Ok(())
}

#[quickcheck]
fn qc_bb_is_really_correct(inst: Instance) {
    assert_eq!(inst.branch_and_bound().cost, inst.brute_force().cost);
}

#[quickcheck]
fn qc_dp_matches_bb(inst: Instance) {
    assert!(inst.branch_and_bound().cost <= inst.dynamic_programming_w().cost);
```

```
    }

    #[quickcheck]
    fn qc_dps_match(inst: Instance) {
        assert_eq!(inst.dynamic_programming_w().cost, inst.dynamic_programming_c().cost);
    }

    #[quickcheck]
    fn qc_greedy_is_valid(inst: Instance) {
        inst.greedy().assert_valid();
        inst.greedy_redux().assert_valid();
    }
}
```