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

Pro provedení měření výkonu programu jsem využil nástroje Hyperfine.

```
uname -a
./cpufetch --logo-short --color ibm
mkdir -p docs/measurements/
cd solver
hyperfine --export-json ../docs/bench.json \
          --parameter-list n 4,10,15 \
          --parameter-list set NK,ZKC,ZKW \
          --parameter-list alg bb,dpc,dpw,fptas1,fptas2,greedy,redux \
          --min-runs 4 \
          --style color \
          'cargo run --release -- {alg} \
           < ds/{set}{n}_inst.dat \
           > ../docs/measurements/{alg}-{set}{n}.txt' 2>&1 \
    | fold -w 120 -s
```

Měření ze spuštění Hyperfine jsou uložena v souboru `docs/bench.json`, který následně zpracujeme do tabulky níže.

```
echo "alg.,sada,\$n\$,průměr,\$\pm \sigma\$,minimum,medián,maximum" > docs/bench.csv
jq -r \
   '.[] | .[] | [.parameters.alg, .parameters.set, .parameters.n
                , ([.mean, .stddev, .min, .median, .max]
                   | map("**" + (100000 * . + 0.5
                         | floor
                         | . / 100
                         | tostring
                         | if test("\\.") then sub("\\."; "**.") else . + "**" end
                         ) + " ms"
                        )
                  )
                ] | flatten | @csv' \
   docs/bench.json \
>> docs/bench.csv
echo ""
```

Table 1: Měření výkonu pro různé kombinace velikosti instancí problému (n) a zvoleného algoritmu.

alg.	sada	n	průměr	$\pm\sigma$	minimum	medián	maximum
bb	NK	4	**120**.47 ms	**2**.83 ms	**116**.02 ms	**119**.92 ms	**127**.09 ms
bb	NK	10	**122**.56 ms	**3**.2 ms	**116**.1 ms	**122**.75 ms	**130**.1 ms
bb	NK	15	**149**.17 ms	**2**.32 ms	**145**.64 ms	**148**.54 ms	**153**.27 ms
bb	ZKC	4	**121**.63 ms	**4**.18 ms	**116**.55 ms	**120**.67 ms	**135**.87 ms
bb	ZKC	10	**126**.09 ms	**3**.19 ms	**120**.58 ms	**125**.79 ms	**134**.02 ms
bb	ZKC	15	**256**.6 ms	**9**.5 ms	**248** ms	**252**.29 ms	**273**.65 ms
bb	ZKW	4	**121**.37 ms	**4**.27 ms	**116**.73 ms	**119**.27 ms	**131**.09 ms
bb	ZKW	10	**121**.02 ms	**4**.62 ms	**116**.07 ms	**119**.48 ms	**131**.95 ms
bb	ZKW	15	**120**.99 ms	**3**.7 ms	**116**.34 ms	**120**.29 ms	**133**.57 ms
dpc	NK	4	**510**.4 ms	**8**.02 ms	**503**.39 ms	**505**.65 ms	**519**.87 ms
dpc	NK	10	**2877**.62 ms	**9**.65 ms	**2867**.37 ms	**2877**.4 ms	**2888**.3 ms
dpc	NK	15	**6779**.01 ms	**19**.27 ms	**6756**.01 ms	**6778**.44 ms	**6803**.13 ms
dpc	ZKC	4	**501**.24 ms	**5**.17 ms	**494**.89 ms	**500**.61 ms	**507**.1 ms
dpc	ZKC	10	**2901**.21 ms	**15**.87 ms	**2885**.34 ms	**2898**.45 ms	**2922**.59 ms
dpc	ZKC	15	**6827**.9 ms	**11**.67 ms	**6813**.07 ms	**6829**.33 ms	**6839**.86 ms
dpc	ZKW	4	**2588**.47 ms	**9**.62 ms	**2579**.91 ms	**2585**.87 ms	**2602**.21 ms
dpc	ZKW	10	**5530**.76 ms	**38**.19 ms	**5497**.65 ms	**5520**.09 ms	**5585**.21 ms
dpc	ZKW	15	**4833**.57 ms	**44**.19 ms	**4783**.6 ms	**4833**.31 ms	**4884**.07 ms
dpw	NK	4	**127**.02 ms	**5**.2 ms	**121**.34 ms	**126**.31 ms	**143**.19 ms
dpw	NK	10	**179**.85 ms	**4**.58 ms	**173**.43 ms	**179**.82 ms	**189**.81 ms
dpw	NK	15	**267**.01 ms	**3**.88 ms	**260**.82 ms	**267**.11 ms	**272**.67 ms
dpw	ZKC	4	**130**.79 ms	**4**.2 ms	**124**.14 ms	**130**.53 ms	**141**.58 ms
dpw	ZKC	10	**200**.33 ms	**3**.29 ms	**195**.35 ms	**200**.31 ms	**206**.38 ms
dpw	ZKC	15	**313**.02 ms	**8**.16 ms	**303**.06 ms	**313**.4 ms	**330**.47 ms
dpw	ZKW	4	**132** ms	**3**.27 ms	**128**.14 ms	**131**.02 ms	**140**.86 ms
dpw	ZKW	10	**131**.16 ms	**5**.24 ms	**121**.28 ms	**131**.69 ms	**144**.85 ms
dpw	ZKW	15	**128**.31 ms	**4**.69 ms	**121**.56 ms	**126**.45 ms	**137**.82 ms
fptas1	NK	4	**127**.46 ms	**3**.17 ms	**123**.26 ms	**126**.73 ms	**134**.8 ms

alg.	sada	n	průměr	$\pm\sigma$	minimum	medián	maximum
fptas1	NK	10	**239**.16 ms	**10**.09 ms	**221**.09 ms	**238**.12 ms	**254**.38 ms
fptas1	NK	15	**525**.7 ms	**9**.15 ms	**513**.77 ms	**530**.2 ms	**533**.71 ms
fptas1	ZKC	4	**127**.06 ms	**6**.04 ms	**117**.15 ms	**127**.51 ms	**143**.9 ms
fptas1	ZKC	10	**238**.52 ms	**7**.97 ms	**226**.63 ms	**239**.07 ms	**249**.63 ms
fptas1	ZKC	15	**549**.06 ms	**10**.45 ms	**536**.04 ms	**551**.35 ms	**558**.91 ms
fptas1	ZKW	4	**138**.94 ms	**5**.81 ms	**131**.75 ms	**138**.18 ms	**159**.88 ms
fptas1	ZKW	10	**196**.95 ms	**3**.84 ms	**190**.95 ms	**196**.36 ms	**202**.6 ms
fptas1	ZKW	15	**250**.27 ms	**4**.68 ms	**244**.9 ms	**250**.24 ms	**258**.46 ms
fptas2	NK	4	**189**.16 ms	**4**.27 ms	**184**.93 ms	**187**.57 ms	**200**.85 ms
fptas2	NK	10	**1399**.53 ms	**7**.03 ms	**1393**.43 ms	**1397**.99 ms	**1408**.71 ms
fptas2	NK	15	**4557**.56 ms	**17**.11 ms	**4539**.41 ms	**4559**.11 ms	**4572**.62 ms
fptas2	ZKC	4	**190**.95 ms	**6**.61 ms	**183**.42 ms	**188**.71 ms	**208**.05 ms
fptas2	ZKC	10	**1424**.34 ms	**31**.38 ms	**1397**.45 ms	**1416**.17 ms	**1467**.56 ms
fptas2	ZKC	15	**4525**.8 ms	**54**.94 ms	**4443**.58 ms	**4551**.65 ms	**4556**.3 ms
fptas2	ZKW	4	**258**.3 ms	**6**.15 ms	**252**.25 ms	**255**.99 ms	**274**.74 ms
fptas2	ZKW	10	**897**.05 ms	**6**.86 ms	**890**.48 ms	**896**.67 ms	**904**.39 ms
fptas2	ZKW	15	**1490**.83 ms	**9**.35 ms	**1477**.96 ms	**1492**.56 ms	**1500**.22 ms
greedy	NK	4	**123**.21 ms	**2**.6 ms	**117**.48 ms	**122**.7 ms	**128**.42 ms
greedy	NK	10	**123**.38 ms	**2**.58 ms	**120**.13 ms	**122**.97 ms	**130**.42 ms
greedy	NK	15	**125**.41 ms	**2**.14 ms	**122**.03 ms	**125**.09 ms	**130**.54 ms
greedy	ZKC	4	**122**.01 ms	**1**.94 ms	**119**.43 ms	**121**.75 ms	**125**.89 ms
greedy	ZKC	10	**124**.17 ms	**2**.63 ms	**119**.05 ms	**123**.91 ms	**129**.83 ms
greedy	ZKC	15	**123**.93 ms	**2**.57 ms	**118**.56 ms	**123**.89 ms	**128**.45 ms
greedy	ZKW	4	**126**.53 ms	**4**.53 ms	**121**.57 ms	**125**.12 ms	**139**.37 ms
greedy	ZKW	10	**123**.11 ms	**2**.39 ms	**118**.98 ms	**122**.58 ms	**127**.32 ms
greedy	ZKW	15	**124**.7 ms	**5**.1 ms	**118**.7 ms	**124**.11 ms	**138**.32 ms
redux	NK	4	**124**.7 ms	**5**.1 ms	**119**.06 ms	**123**.56 ms	**137**.57 ms
redux	NK	10	**127**.22 ms	**7**.41 ms	**119**.01 ms	**125**.52 ms	**143**.53 ms
redux	NK	15	**124**.26 ms	**3**.46 ms	**121**.03 ms	**123**.59 ms	**137** ms
redux	ZKC	4	**123**.34 ms	**3**.46 ms	**117**.55 ms	**123**.49 ms	**131**.15 ms
redux	ZKC	10	**124**.94 ms	**5**.57 ms	**119**.37 ms	**122**.91 ms	**141**.71 ms
redux	ZKC	15	**123**.71 ms	**2**.47 ms	**119**.24 ms	**123**.49 ms	**128**.38 ms
redux	ZKW	4	**124**.2 ms	**2**.7 ms	**120**.87 ms	**124**.21 ms	**130**.74 ms
redux	ZKW	10	**123**.67 ms	**3**.25 ms	**118**.59 ms	**123**.59 ms	**131**.48 ms
redux	ZKW	15	**122**.8 ms	**2**.14 ms	**120**.02 ms	**122**.05 ms	**127**.49 ms

Srovnání algoritmů

```
<<preprocessing>>

<<performance-chart>>

<<histogram>>
```

```
import matplotlib.pyplot as plt
import pandas as pd
from pandas.core.tools.numeric import to_numeric

bench = pd.read_csv("docs/bench.csv", dtype = "string")
bench.rename({
```

```
        "alg.": "alg",
        "$n$": "n",
        "sada": "set",
        "průměr": "avg",
        "$\pm \sigma$": "sigma",
        "medián": "median",
        "minimum": "min",
        "maximum": "max",
    },
    inplace = True,
    errors  = "raise",
    axis    = 1,
)

numeric_columns = ["n", "avg", "sigma", "min", "median", "max"]
bench[numeric_columns] = bench[numeric_columns].apply(lambda c:
    c.apply(lambda x:
        to_numeric(x.replace("**", "").replace(" ms", ""))
    )
)

# Create a figure and a set of subplots.
fig, ax = plt.subplots(figsize = (11, 6))
labels = { "bf"    : "Hrubá síla"
         , "bb"    : "Branch & bound"
         , "dpw"   : "Dynamické programování - podle váhy"
         , "dpc"   : "Dynamické programování - podle ceny"
         , "fptas1": "FPTAS eps = 0.1"
         , "fptas2": "FPTAS eps = 0.01"
         , "greedy": "Hladový přístup"
         , "redux" : "Hladový přístup - redux"
         }

# Group the dataframe by alg and create a line for each group.
for name, group in bench.groupby(["alg", "set"]):
    (x, y, sigma) = (group["n"], group["avg"], group["sigma"])
    ax.plot(x, y, label = labels[name[0]] + " na sadě " + name[1])
    ax.fill_between(x, y + sigma, y - sigma, alpha = 0.3)

# Axis metadata: ticks, scaling, margins, and the legend
plt.xticks(bench["n"])
ax.set_yscale("log", base = 10)
ax.margins(0.05, 0.1)
ax.legend(loc="upper left")

# Reverse the legend
handles, labels = plt.gca().get_legend_handles_labels()
order = range(len(labels) - 1, -1, -1)
plt.legend([handles[idx] for idx in order],[labels[idx] for idx in order])

plt.savefig("docs/assets/graph.svg")
```

```
import os
```

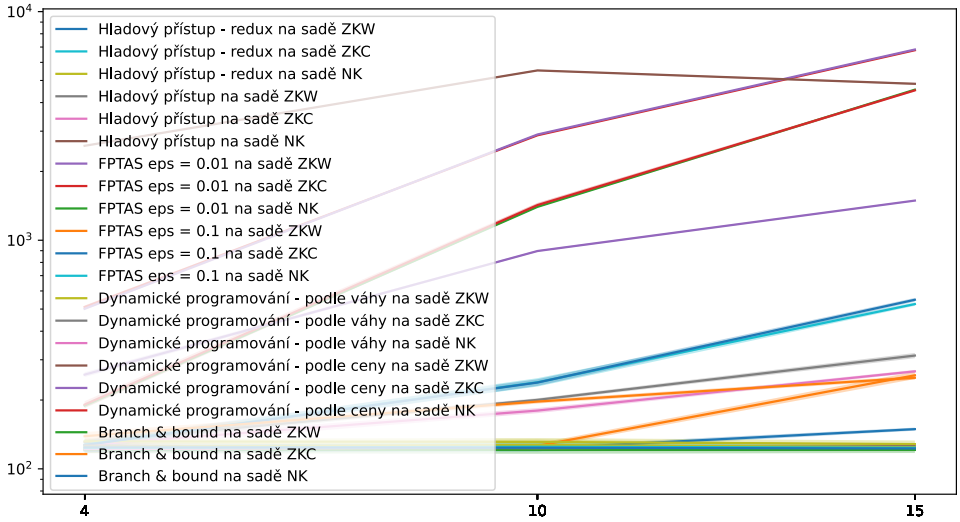

Figure 1: Závislost doby běhu na počtu předmětů. Částečně průhledná oblast značí směrodatnou odchylku (σ).

```
# Load the data
data = []

for filename in os.listdir('docs/measurements'):
    if filename.endswith(".txt"):
        alg = filename[:-4]
        with open('docs/measurements/' + filename) as f:
            for line in f:
                data.append({'alg': alg, 'n': int(line)})

df = pd.DataFrame(data)

# Plot the histograms

for alg in df.alg.unique():
    plt.figure()
    plt.xlabel('Počet konfigurací')
    plt.ylabel('Četnost výskytu')
    plt.hist(df[df.alg == alg].n, color = 'tab:blue' if alg[-3] == 'N' else 'orange', bins = 20)
    plt.xlim(xmin = 0)
    plt.savefig('docs/assets/histogram-' + alg + '.svg')
    plt.close()
```

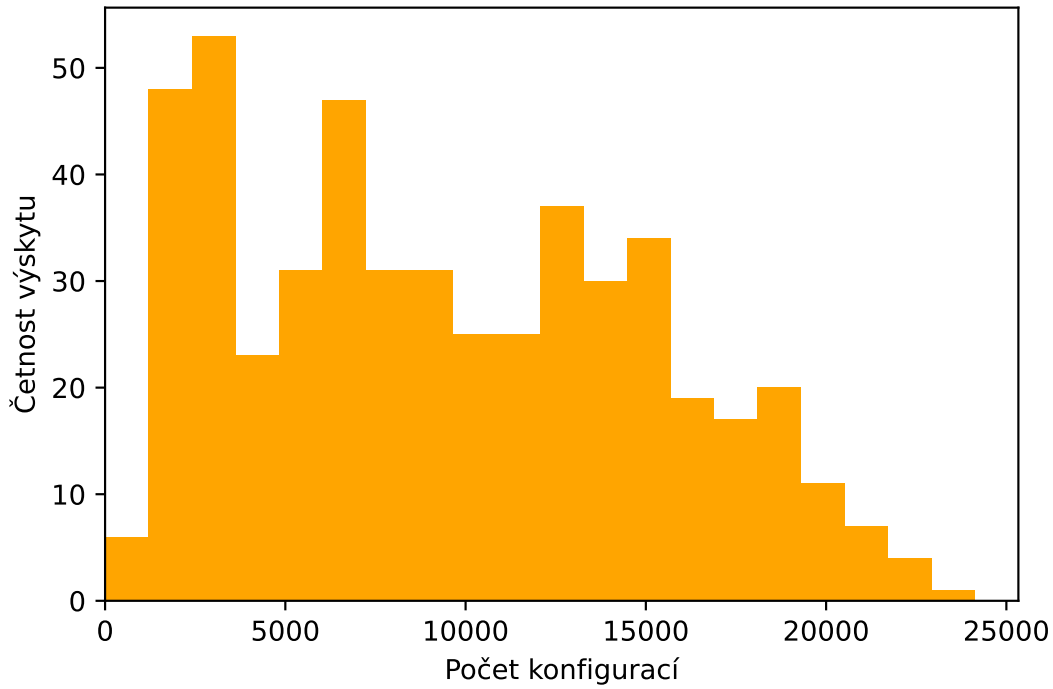

Figure 2: Sada NR: Redux pro $n = 15$

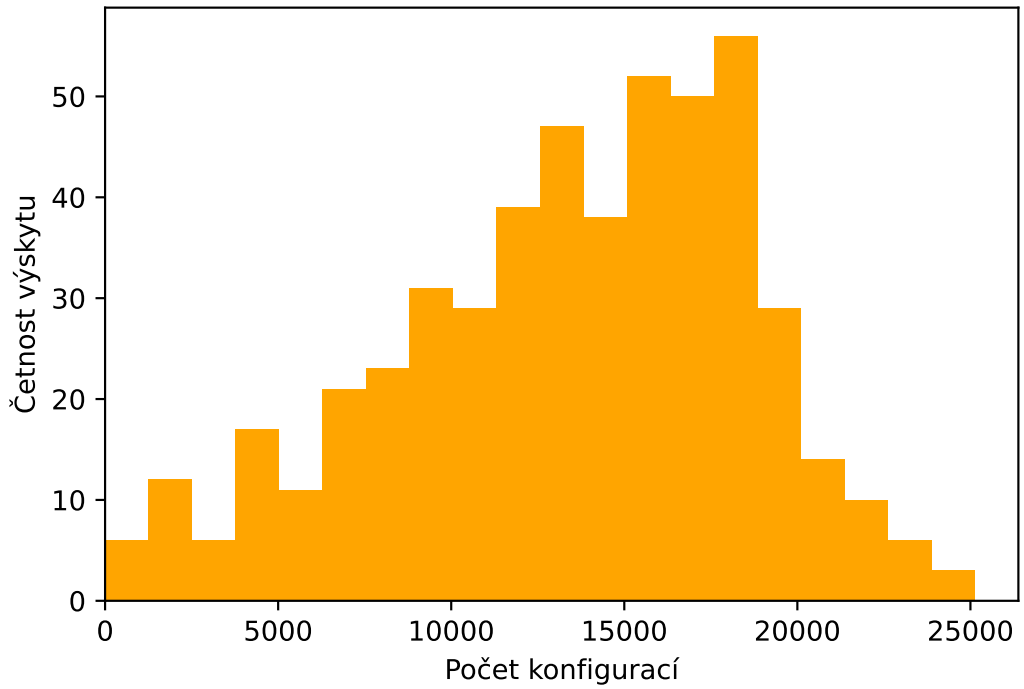

Figure 3: Sada NR: Metoda větví a hranic pro $n = 15$

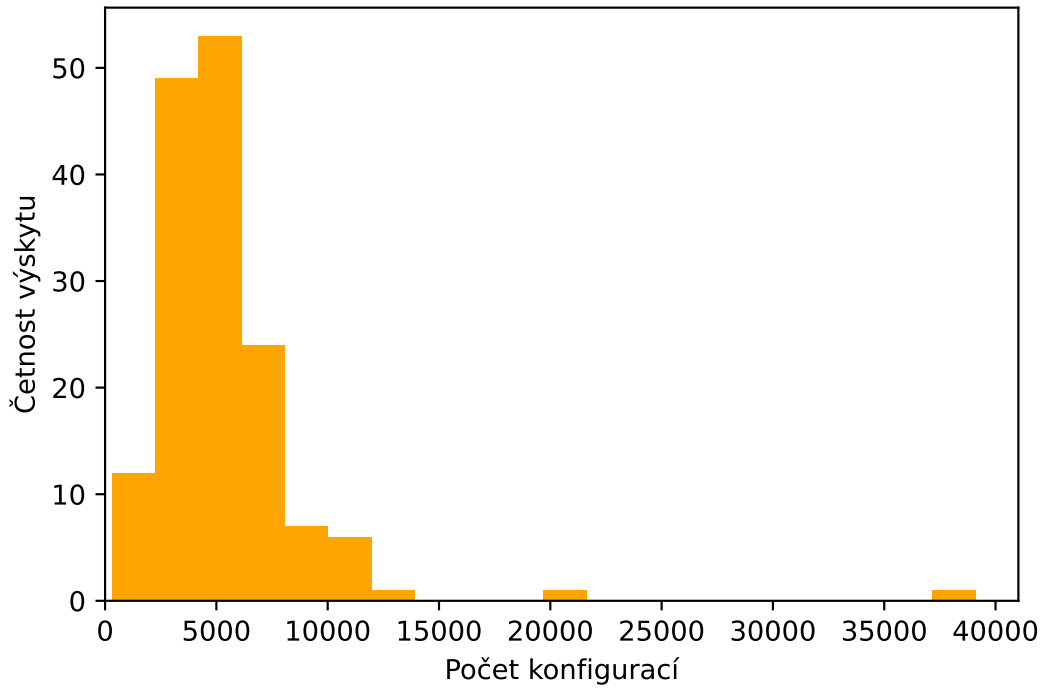

Figure 4: Sada ZR: Redux pro $n = 15$

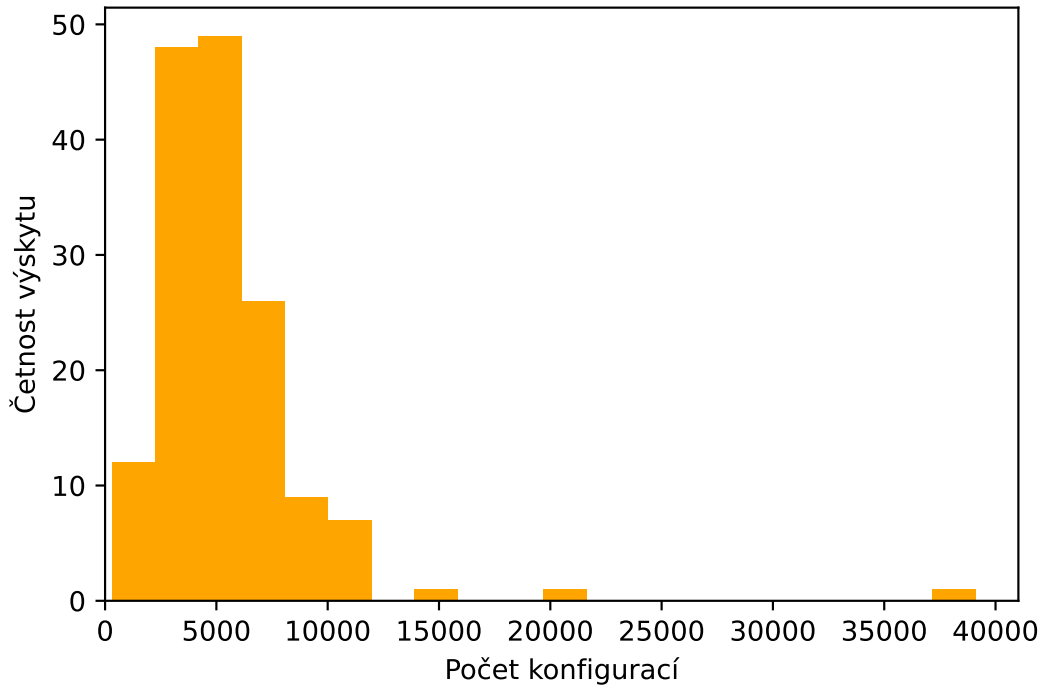

Figure 5: Sada ZR: Metoda větví a hranic pro $n = 15$

Analýza

Vyhovují nejhorší případy očekávané závislosti? Ano. Jak ukazují měření, s rostoucím počtem předmětů v batohu počet konfigurací i skutečný CPU čas velmi rychle roste. Pro $n < 15$ můžeme pozorovat jisté fluktuace doby běhu, pro větší instance už ale není pochyb.

Závisí střední hodnota výpočetní závislosti na sadě instancí? Pro hrubou sílu není znát velký rozdíl, ale rozdíly metody větví a hranic jsou mezi sadami dobře vidět z histogramů v předchozí podsekci. Metoda větví a hranic si k rychlému ukončení dopomáhá součtem cen dosud nepřidaných předmětů – strom rekurzivních volání se zařízne, pokud nepřidané předměty nemohou dosáhnout ceny nejlepšího známého řešení. Aby tato podmínka výpočet urychlila, měly by se lehké, cenné předměty nacházet především na začátku seznamu. Zdá se, že sada ZR obsahuje méně takto vhodných instancí. Algoritmus bych chtěl do budoucna vylepšit předzpracováním v podobě seřazení seznamu předmětů.

Implementace

Program začíná definicí datové struktury reprezentující instanci problému batohu.

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

fn main() -> Result<()> {
    let algorithms = get_algorithms();

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

    loop {
        match parse_line(&mut stdin().lock())?.as_ref().map(alg) {
            Some(Solution { cost, .. }) => println!("{}", cost),
            None => return Ok(())
        }
    }
}

<<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.

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

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

#[derive(Debug, PartialEq, Eq, Clone)]
struct OptimalSolution { id: i32, 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.

```
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 { c as f64 / w as f64 }
    let permutation = sort_by(
        &(items)[..],
        |a, b|
            ratio(*a)
            .partial_cmp(&ratio(*b))
            .unwrap()
            .reverse() // max value 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 in 0..items.len() {
        let (weight, _cost) = items[i];
        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 in 0..items.len() {
        let (_weight, cost) = items[i];
        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

Tento úvodní problém byl příležitostí připravit si technické zázemí pro nadcházející úkoly. Implementace, které odevzdávám, se značně spoléhají na bezpečí typového systému jazyka Rust. Čas ukáže, jestli to usnadní jejich další rozšiřování a obohacování. Zadání jsem se pokusil splnit v celém rozsahu, ale neměl jsem už čas implementovat ořezávání s pomocí fragmentální varianty problému batohu v metodě větví a hranic.

Appendix

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

```
use std::{cmp, cmp::max, collections::BTreeMap, io::{stdin, BufRead}, str::FromStr};
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>>

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}))
}

#[cfg(test)]
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}))
}
```

Výběr algoritmu je řízen argumentem předaným na příkazové řádce. Příslušnou funkci 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))
}
```

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))
    }
}
```

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};
    use std::{fs::{read_dir, ReadDir, DirEntry, File}, iter::Filter, io::BufReader, collections::Hash

    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);
    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);
}

fn load_solutions() -> 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("NK") &&
            name.ends_with("_sol.dat")
        }).is_some());

    for file in files {
        let file = file?;
        let n = file.file_name().into_string().unwrap()[2..].split('_').nth(0).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)
}

#[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()?;
    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()?;
    // 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();
}

#[test]
```

```rust
fn fptas_is_within_bounds() -> Result<()> {
    let opts = load_solutions()?;
    for eps in [0.1, 0.01] {
        for file in list_input_files()? {
            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(())
}

type FilterClosure = fn(&std::result::Result<DirEntry, std::io::Error>) -> bool;
fn list_input_files() -> Result<Filter<ReadDir, FilterClosure>> {
    let f: FilterClosure = |res| 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 NK,
        file_name.starts_with("NK") &&
        // ... continue with an integer between 0 and 15,
        file_name[2..]
        .split('_').nth(0).unwrap().parse::<u32>().ok()
        .filter(|n| (0..9).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))
}

#[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);
    }
}
```