# NI-KOP – úkol 5

Ondřej Kvapil

## Kombinatorická optimalizace: problém vážené splnitelnosti booleovské formule

### Zadání

#### Pokyny

Problém řešte některou z pokročilých heuristik:

- simulované ochlazování
- genetický algoritmus
- tabu prohledávání

Po nasazení heuristiky ověřte její vlastnosti experimentálním vyhodnocením, které přesvědčivě doloží, jakou třídu (rozsah, velikosti...) instancí heuristika zpracovává. Zejména v případě použití nestandardních, např. originálních technik doložte jejich účinnost experimentálně (což vyloučí případné diskuse o jejich vhodnosti).

Zpráva by měla dokládat Váš racionální přístup k řešení problému, tedy celý pracovní postup. Ve zprávě prosím také popište obě fáze nasazení heuristiky, jak nastavení, (white box fáze), tak závěrečné vyhodnocení heuristiky (black box fáze). Prosím používejte definované formáty pro instance a řešení, usnadníte tak lepší přizpůsobení zkušebních instancí.

#### Hodnocení

Tato úloha je součástí hodnocení zkoušky - až 30 bodů ze 100 za předmět celkem. Práce by měla doložit Vaši schopnost nasadit pokročilé heuristiky na netriviální optimalizační problém. Nasazená heuristika by měla zpracovávat rozumně široké spektrum instancí s rozumnou chybou. Co je "rozumně", bychom se měli dočíst v závěru Vaší práce.

V hodnocení je kladen důraz na racionální postup celé práce. Pokud postup vyhovuje, méně uspokojivé výsledky heuristiky příliš nevadí, vzhledem k tomu, že řešený problém (jak jistě víte) patří k nejtěžším ve třídě NPO. Proto potřebujeme znát jak pracovní postup ve white box fázi, tak výsledky a závěr black box fáze.

Hodnocení je rozděleno do tří kategorií:

- Algoritmus a implementace (5 pt.)
  - Byly použity techniky (algoritmy, datové struktury) adekvátní problému?
  - Byly použity pokročilé techniky? (např. adaptační mechanismy)
  - Jsou některé postupy originálním přínosem autora?
- Nastavení heuristiky (13 pt.)
  - Jakou metodou autor hledal nastavení parametrů?
  - Jak byly plánovány experimenty a jaké byly jejich otázky?

  - Jestliže byl proveden faktorový návrh (což příliš nedoporučujeme), jak kompletní byl (změna vždy jen jednoho parametru nestačí)?
  - Na jak velkých instancích je heuristika schopna pracovat?
  - Jestliže práce heuristiky není uspokojivá, jak systematické byly snahy autora zjednat nápravu?
- Experimentální vyhodnocení heuristiky (12 pt.)
  - Jak dalece jsou závěry vyhodnocení doloženy experimentálně?
  - Je interpretace experimentů přesvědčivá?
  - Pokud je algoritmus randomizovaný, byla tato skutečnost vzata v úvahu při plánování experimentů?
  - Je možno z experimentů usoudit na iterativní sílu heuristiky?
  - Byly nestandardní postupy experimentálně porovnány se standardními?
  - Jsou výsledky experimentů srozumitelně prezentovány (grafy, tabulky, statistické metody)?

Práce bez experimentální části nemůže být přijata k hodnocení.

**Instance**

- SAT instance lze generovat náhodně. Klíčovým parametrem je poměr počtu klauzulí k počtu proměnných pro 3-SAT (viz ai-phys1.pdf - doporučujeme). Váhy lze generovat náhodně. V takovém případě je vhodné prokázat, že instance, kde všechny váhy byly vynásobeny velkým číslem, jsou zpracovávány stejně úspěšně.
- Lze vyjít z DIMACS SAT instancí. Nemají váhy, jejich generování viz výše.Tyto instance jsou na hraně fázového přechodu, jsou tedy značně obtížné. Obtížnost můžete snížit zkrácením (vynechání klauzulí).
- Připravili jsme sady zkušebních instancí. Vycházejí z instancí SATLIB, jsou ale zkráceny tak, aby měly co nejvíce řešení (počty řešení přikládáme). Váhy nejsou náhodné, mají ale náhodnou složku. Za upozornění na chyby, nekonzistence atd. budeme vděčni.
  - `wuf-M` a `wuf-N`: Váhy by měly podporovat nalezení řešení. Heuristika, která řeší určitou instanci v sadě `wuf-M`, by měla řešit odpovídající instanci v sadě `wuf-N` stejně snadno. Vzhledem k počtu řešení jednotlivých instací (až 108), není možné efektivně najít optimální řešení
  - `wuf-Q` a `wuf-R`: Jako výše, váhy ale vytvářejí (mírně) zavádějící úlohu.
  - `wuf-A`: z dílny prof. Zlomyslného. Instance vycházejí z nezkrácených (nebo jen mírně zkrácených) instancí, takže jsou obtížné. Váhy vytvářejí zavádějící úlohu. Nicméně, malý počet řešení dovoluje ve většině případů uvést optimální řešení.

## 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

Stejně jako v předchozí úloze jsem se ani tentokrát s měřením výkonu nespoléhal na existující Rust knihovny a namísto toho provedl měření v Pythonu.

```
uname -a
./cpufetch --logo-short --color ibm
```

### White box: průzkum chování algoritmu

Průzkumná fáze práce má za cíl odhalit vztahy mezi jednotlivými parametry algoritmu a najít pro ně vhodné hodnoty. Výchozí parametry jsou vidět v kódu níže, modifikátor teploty je 0.7 a maximální počet iterací je 8000. Modifikátor teploty je koeficient, kterým se násobí cena řešení heuristické metody greedy redux. Toto číslo určuje počáteční teplotu.

Průzkum jsem provedl na prvních dvaceti instancích sady `NK35`.

```
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
import numpy as np
import scipy.stats as st
import json
import os
import time
from pandas.core.tools.numeric import to_numeric
from subprocess import run, PIPE
from itertools import product, chain
import textwrap as tr
```

Skript `charts.py` je zodpovědný nejen za vykreslování grafů, ale také za načítání vstupních instancí, spouštění řešiče a měření času, který problém řešiči zabere.

```
<<python-imports>>

<<performance-chart>>
```

Výkon každého algoritmu je na instancích měřen jednotlivě, tj. vstupní soubor rozdělíme na řádky a řešiči předáme jedinou instanci.

```
# plot the measurements

figsize = (14, 8)

show_progress = os.environ.get("JUPYTER") == None

# adapted from https://stackoverflow.com/questions/3173320/text-progress-bar-in-the-console
def progress_bar(iteration, total, length = 60):
    if not show_progress:
        return
    percent = ("{0:.1f}").format(100 * (iteration / float(total)))
    filledLength = int(length * iteration // total)
    bar = '=' * filledLength + ' ' * (length - filledLength)
    print(f'\r[{bar}] {percent}%', end = "\r")
    if iteration == total:
        print()

def invoke_solver(input, cfg):
    solver = run(
        [
```

```
            "target/release/main",
            "sa",
            str(cfg["max_iterations"]),
            str(cfg["scaling_factor"]),
            str(cfg["temperature_modifier"]),
            str(cfg["equilibrium_width"]),
        ],
        stdout = PIPE,
        input = input,
        encoding = "ascii",
        cwd = "solver/"
    )
    if solver.returncode != 0:
        print(solver)
        raise Exception("solver failed")

    lines = solver.stdout.split("\n")
    [_, time, _] = lines[-3].split()
    [cost, err]  = lines[-2].split()
    cost_temperature_progression = [list(map(float, entry.split())) for entry in lines[:-4]]
    return (float(time), float(cost), float(err), cost_temperature_progression)

def dataset(id, **kwargs):
    params = dict({
        # defaults
        "id": [id],
        "precise_plot": [True],
        "n_instances": [15],
        "max_iterations": [8000],
        "scaling_factor": [0.996],
        "temperature_modifier": [0.7],
        "equilibrium_width": [10],
    }, **kwargs)

    key_order = [k for k in params]
    cartesian = list(product(
        *[params[key] for key in key_order]
    ))

    return {
        key: [row[key_order.index(key)] for row in cartesian] for key in params
    }

def merge_datasets(*dss):
    return {
        k: list(chain(*(ds[k] for ds in dss)))
        for k in dss[0]
    }

configs = merge_datasets(dataset(
    "scaling_factor_exploration",
    scaling_factor = [0.85, 0.9, 0.95, 0.99, 0.992, 0.994, 0.996, 0.997, 0.998, 0.999],
```

```
), dataset(
    "temperature modifier exploration",
    n_instances = [30],
    temperature_modifier = [0.0001, 0.01, 1, 100, 10000],
), dataset(
    "equilibrium width exploration",
    n_instances = [40],
    precise_plot = [False],
    equilibrium_width = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100],
), dataset(
    "black box",
    precise_plot = [False],
    n_instances = [500],
))

# load the input
input = None
with open("solver/ds/NK35_inst.dat", "r") as f:
    input = f.read()

errors = []
cfgs = [dict(zip(configs, v)) for v in zip(*configs.values())]
iteration = 0
total = sum([cfg["n_instances"] for cfg in cfgs])

for config in cfgs:
    if show_progress:
        print(end = "\033[2K")
    print(config)
    progress_bar(iteration, total)

    params = "-".join([str(v) for _, v in config.items()])
    for instance in input.split("\n")[:config["n_instances"]]:
        id = instance.split()[0]
        (t, cost, err, cost_temperature_progression) = invoke_solver(instance, config)
        errors.append(dict(config, error = err))

        if config["precise_plot"]:
            # plot the cost / temperature progression:
            # we have two line graphs in a single plot
            # the x axis is just the index in the list

            plt.style.use("dark_background")
            fig, ax = plt.subplots(figsize = figsize)
            for (i, label) in zip(range(42), ["cost", "best cost", "temperature"]):
                ax.plot(
                    range(len(cost_temperature_progression)),
                    [entry[i] for entry in cost_temperature_progression],
                    label = label,
                )
            ax.set_xlabel("iteration")
            ax.set_title(f"instance {id} with error {err}")
```

```
            ax.legend(loc = "lower right")

            plt.savefig(f"docs/assets/whitebox-{params}-{id}.svg")
            plt.close()

        iteration = iteration + 1
        progress_bar(iteration, total)

data = pd.DataFrame(errors)
def ridgeline(id, title, col, filename, x_label = "Chyba oproti optimálnímu řešení [%]"):
    df = data[data["id"] == id]
    series = df.groupby(col)["error"].mean()
    df["mean error"] = df[col].map(series)

    # plot the error distributions for each value of col
    plt.style.use("default")
    sns.set_theme(style = "white", rc = {"axes.facecolor": (0, 0, 0, 0)})
    pal = sns.color_palette("crest", n_colors = len(df[col].unique()))

    # set up the layout
    g = sns.FacetGrid(
        df,
        row = col,
        hue = "mean error",
        palette = pal,
        height = 0.75,
        aspect = 15,
    )
    plt.xlim(-0.1, 1.0)
    # distributions
    g.map(sns.kdeplot, "error", clip = (-0.1, 1.0), bw_adjust = 1, clip_on = False, fill = True, alph
    # contours
    g.map(sns.kdeplot, "error", clip = (-0.1, 1.0), bw_adjust = 1, clip_on = False, color = "w", lw =
    # horizontal lines
    g.map(plt.axhline, y = 0, lw = 2, clip_on = False)
    # overlap
    g.fig.subplots_adjust(hspace = -0.3)

    for i, ax in enumerate(g.axes.flat):
        ax.text(-0.125, 5, df[col].unique()[i],
                fontsize = 15, color = ax.lines[-1].get_color(), va = "baseline")

    # remove titles, y ticks, spines
    g.set_titles("")
    g.set(yticks = [])
    g.despine(left = True, bottom = True)
    g.fig.suptitle(title, fontsize = 20, ha = "right")
    for ax in g.axes.flat:
        ax.xaxis.set_major_formatter(lambda x, pos: f"{x * 100:.0f}")
    g.set_xlabels(x_label)
    g.set_ylabels("")
```

```
    g.savefig(f"docs/assets/{filename}")
    plt.close()

ridgeline(
    "scaling_factor_exploration",
    "Vliv koeficientu chlazení na hustotu chyb",
    "scaling_factor",
    "whitebox-error-distributions.svg",
)

ridgeline(
    "temperature modifier exploration",
    "Vliv koeficientu počáteční teploty na hustotu chyb",
    "temperature_modifier",
    "whitebox-error-distributions-temperature.svg",
)

ridgeline(
    "equilibrium width exploration",
    "Vliv šířky ekvilibria na hustotu chyb",
    "equilibrium_width",
    "whitebox-error-distributions-equilibrium-width.svg",
)

# plot the error distribution
sns.kdeplot(
    data = data[data["id"] == "black box"],
    x = "error",
)
plt.savefig("docs/assets/blackbox-error-distribution.svg")
```

Koeficient chlazení má na průběh hledání zásadní vliv. Přehled různých nastavení tohoto parametru je vidět v grafu níže.

Vliv koeficientu chlazení na hustotu chyb

Rychlé chlazení vede k nedostatečné diversifikaci řešení. Algoritmus nemá možnost projít skrz řešení nízké ceny aby se dostal z oblasti lokálního minima. Protože ukončovací podmínka závisí na teplotě, tento postup nevyzkouší mnoho možností a už po pár desítkách iterací skončí – s vysokou chybou.

Nedostatečná diversifikace rychlého chlazení

U velmi pomalého chlazení dochází k předčasnému vyčerpání limitu iterací. Algoritmus skončí dříve, než teplota klesne natolik, aby začala intensifikace.

Nedostatečná intensifikace pomalého chlazení

Koeficient 0.996 je ideální volbou pro toto konkrétní nastavení ostatních parametrů. Řešení se blíží optimu s maximální chybou $\approx 2.43\%$ a průměrnou $\approx 0.36\%$.

Fáze diversifikace i intensifikace ve vhodném poměru

Vliv parametrů počáteční teploty a šířky ekvilibria na hustotu chyb jsem změřil podobným způsobem, tyto parametry zřejmě kvalitu řešení zdaleka neovlivňují tolik.

Pro připomenutí, koeficient počáteční teploty násobí cenu řešení greedy redux, toto číslo udává počáteční teplotu.

Vliv koeficientu počáteční teploty na hustotu chyb

Šířka ekvilibria určuje počet iterací při kterých algoritmus setrvá v jedné teplotě, nikoliv však jak rychle se teplota mění.

Vliv šířky ekvilibria na hustotu chyb

#### Black box: vyhodnocení hustoty chyb

Pro black box vyhodnocení jsem použil celou sadu NK35 a změřil hustotu chyb.

Hustota chyb přes 500 instancí

## 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, Option<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.get(&(inst.items.len() as u32, inst.id));
                let error = optimal.map(|opt| 1.0 - sln.cost as f64 / opt.cost as f64);
                results.push((sln.cost, error))
            },
            None => return Ok(results)
        }
    }
}

pub fn load_instances<T>(stream: &mut T) -> Result<Vec<Instance>>
where T: BufRead {
    let mut instances = vec![];
    loop {
        match parse_line(stream)? {
            Some(inst) => instances.push(inst),
            None => return Ok(instances)
```

```
        }
    }
}

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>>

trait IteratorRandomWeighted: Iterator + Sized + Clone {
    fn choose_weighted<Rng: ?Sized, W>(&mut self, rng: &mut Rng, f: fn(Self::Item) -> W) -> Option<Se
    where
        Rng: rand::Rng,
        W: for<'a> core::ops::AddAssign<&'a W>
         + rand::distributions::uniform::SampleUniform
         + std::cmp::PartialOrd
         + Default
         + Clone {
        use rand::prelude::*;
        let dist = rand::distributions::WeightedIndex::new(self.clone().map(f)).ok()?;
        self.nth(dist.sample(rng))
    }
}

impl<I> IteratorRandomWeighted for I where I: Iterator + Sized + Clone {}

impl Instance {
    <<solver-dpw>>

    <<solver-dpc>>

    <<solver-fptas>>
```

```
    <<solver-greedy>>

    <<solver-greedy-redux>>

    <<solver-bb>>

    <<solver-bf>>

    <<solver-sa>>
}

<<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> {
        self.set(i, true)
    }

    fn without(mut self, i: usize) -> Solution<'a> {
```

```
        self.set(i, false)
    }

    fn invert(mut self) -> Solution<'a> {
        for i in 0..self.inst.items.len() {
            self.set(i, !self.cfg[i]);
        }
        self
    }

    fn set(&mut self, i: usize, set: bool) -> Solution<'a> {
        let (w, c) = self.inst.items[i];
        let k = if set { 1 } else { -1 };
        if self.cfg[i] != set {
            self.cfg.set(i, set);
            self.weight = (self.weight as i32 + k * w as i32) as u32;
            self.cost   = (self.cost   as i32 + k * c as i32) as u32;
        }
        *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 }
    }

    fn trim<Rng: ?Sized>(&mut self, rng: &mut Rng) where Rng: rand::Rng {
        while self.weight > self.inst.m {
            self.set(rng.gen_range(0..self.inst.items.len()), false);
        }
    }
}
```

### 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

```rust
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** Dynamické programování s rozkladem podle váhy jsem implementoval už v prvním úkolu.

```
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()
}
```

V úkolu 2 přibyla implementace dynamického 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 }
}
```

**Simulované žíhání** Iterativní metoda simulovaného žíhání se od ostatních algoritmů v několika ohledech liší.

Prvně vyžaduje generátor náhodných čísel a konfiguraci jako vstupní parametry a její typ proto neodpovídá ostatním metodám – v souboru `main.rs` je toto třeba ošetřit, připravit generátor a konfiguraci načíst z příkazové řádky.

Navíc je její přístup k prohledávání stavového prostoru založen na přidávání a odebírání předmětů z batohu narozdíl od klasických algoritmů, kde se batoh jen plní. Z tohoto důvodu bylo třeba zobecnit metodu `Solution::with` na `Solution::set`, která libovolné změny v inkluzi předmětů ergonomicky

umožňuje. Má implementace využívá relaxaci – vybraný kandidát pro nový stav může dočasně překročit maximální kapacitu batohu. Před jeho zvážením a randomizovaným přijetím je ovšem kandidát opraven pomocí metody `Solution::trim`, která odebírá náhodné předměty dokud váha řešení neklesne pod danou mez.

```
pub fn simulated_annealing<Rng>(
    &self,
    rng: &mut Rng,
    (max_iterations, scaling_factor, temp_modifier, equilibrium_width): (u32, f64, f64, u8)
) -> Solution
where Rng: rand::Rng + ?Sized {
    let total_cost = self.items.iter().map(|(_, c)| c).sum::<u32>() as f64;
    let mut current = self.greedy_redux();
    let mut best = current;
    let mut temperature = temp_modifier * self.greedy_redux().cost as f64;

    let mut iteration = 0;
    let frozen = |t| t < 0.00001;
    let equilibrium = |i| i % equilibrium_width as u32 == 0;

    while !frozen(temperature) {
        let temp = temperature;
        loop {
            use rand::prelude::IteratorRandom;
            println!("    {} {} {}", current.cost, best.cost, temp);

            let next_sln = (0..self.items.len())
                // construct the set of neighbouring solutions: generate n
                // solutions, each with one item different to the current
                // solution (included or excluded)
                .map(|i| current.clone().set(i, !current.cfg[i]))
                // also add a complete flip as an option
                .chain(std::iter::once(current.invert()))
                // select a neighbour at random
                .choose(rng);
            match next_sln {
                Some(mut new) => {
                    new.trim(rng);
                    let delta = (new.cost as f64 - current.cost as f64) / total_cost;
                    let rnd = rng.gen_range(0.0 .. 1.0);
                    let threshold = (delta / temp).exp();
                    if delta <= 0.0 {
                        // eprintln!(
                        //     "considering {} (current {}),\tdelta {}, rnd {}, temp {},\t(will {} ag
                        //     new.cost,
                        //     current.cost,
                        //     delta,
                        //     rnd,
                        //     temp,
                        //     if rnd < threshold { "succeed" } else { "fail" },
                        //     threshold,
                        // );
```

```
                    }

                    if  delta > 0.0 // the new state is better, accept it right away
                    || // otherwise accept with probability proportional to the cost delta and the te
                        rnd < threshold {
                        current = new;
                        if current.cost > best.cost {
                            best = current;
                        }
                    }
                },
                None => {
                    println!("  early return @ {}, temp {}", iteration, temp);
                    return best
                },
            };
            iteration += 1;
            temperature *= scaling_factor;
            if iteration >= max_iterations {
                println!("  iteration limit exhausted");
                return best
            } else if equilibrium(iteration) { break }
        }
    }

    println!("  frozen @ {}, temp {}", iteration, temperature / scaling_factor);
    best
}
```

## Závěr

Implementoval jsem metodu simulovaného žíhání (alias ochlazování) a prozkoumal jsem její pro a proti. S vhodným nastavením parametrů nachází výrazně lepší řešení než hladové heuristiky a lze využít pro iterativní zlepšování existujících řešení (např. z greedy redux). Počáteční teplota a hranice maximálního počtu iterací navíc dovolují vhodně omezit výpočetní náročnost na úkor kvality nalezených řešení. Je ovšem třeba dát si pozor na interakci jednotlivých parametrů, např. aby nízký koeficient chlazení nevedl k předčasnému ukončení algoritmu.

## 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)]
```

```rust
#[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.

```rust
<<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 z minulého úkolu postavený na knihovně `Criterion.rs` se nachází v souboru níže. Pro měření těchto experimentů ale nebyl použit.

```
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")?;

    enum Either<A, B> { Left(A), Right(B) }
    use  Either::*;

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

    for inst in load_instances(&mut stdin().lock())? {
        use std::time::Instant;
        let (now, sln) = match alg {
            Right(f) => (Instant::now(), f(&inst)),
            Left(cfg) => {
                let mut rng: rand_chacha::ChaCha8Rng = rand::SeedableRng::seed_from_u64(42);
                (Instant::now(), inst.simulated_annealing(&mut rng, cfg))
            },
        };
        println!("took {} ms", now.elapsed().as_millis());
        let optimal = &solutions.get(&(inst.items.len() as u32, inst.id));
        let error = optimal.map(|opt| 1.0 - sln.cost as f64 / opt.cost as f64);
        println!("{} {}", sln.cost, error.map(|e| e.to_string()).unwrap_or_default());
    }
    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(Right(f))
    } else if alg == "sa" { #[allow(clippy::or_fun_call)] { // simulated annealing
        let mut iter = args[2..].iter().map(|str| &str[..]);
        let max_iterations = iter.next().ok_or(anyhow!("not enough params"))?.parse()?;
        let scaling_factor = iter.next().ok_or(anyhow!("not enough params"))?.parse()?;
```

```
        let temp_modifier = iter.next().ok_or(anyhow!("not enough params"))?.parse()?;
        let equilibrium_width = iter.next().ok_or(anyhow!("not enough params"))?.parse()?;
        Ok(Left((max_iterations, scaling_factor, temp_modifier, equilibrium_width)))
    } } else {
        Err(anyhow!("\"{}\" is not a known algorithm", alg))
    }
} else {
    println!(
        "Usage: {} <algorithm>\n\twhere <algorithm> is one of {}\n\tor 'sa' for simulated annealing."
        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};
    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();
            #[allow(clippy::needless_collect)]
            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())
        }
    }
```

```rust
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
            .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);
}

#[ignore]
#[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_else(|| 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]
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();
        }

        #[ignore]
        #[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();
        }
    }
```