NI-KOP – úkol 3

Ondřej Kvapil

Kombinatorická optimalizace: problém batohu

Zadání

U následujících metod pro řešení konstruktivní verze problému batohu

- hrubá síla (pokud z implementace není evidentní úplná necitlivost na vlastnosti instancí)
- metoda větví a hranic, případně ve více variantách
- dynamické programování (dekompozice podle ceny a/nebo hmotnosti). FPTAS algoritmus není nutné testovat, pouze pokud by bylo podezření na jiné chování, než DP
- heuristika - poměr cena/váha

navrhněte a proveďte experimentální vyhodnocení závislosti kvality řešení a výpočetní náročnosti (Vaší implementace) algoritmů na následující parametry instancí:

- poměru kapacity batohu k sumární váze
- korelaci cena/váha
- rozložení vah a granularitě (pozor - zde si uvědomte smysl exponentu granularity)

generovaných generátorem náhodných instancí. Máte-li podezření na další závislosti, modifikujte zdrojový tvar generátoru. V případě, že některá závislost je spolehlivě odvoditelná z instance, můžete toto odvození uvést. Je to časté u hrubé síly. Pozor, o překvapení nebývá nouze, je lépe závislost ověřit. Ověřte robustnost algoritmů.

Pokyny

Pro úsporu času a námahy se experimentální vyhodnocení často dělí na *pilotní* a *detailní* experimenty.

Pilotní experiment nám řekne například jestli sledovaná závislost vůbec existuje, jaké jsou rozsahy veličin, se kterými pracujeme a podobně. Jejich výsledky se většinou nepublikují, slouží k zacílení detailních experimentů. Provádějí se typicky s malým počtem instancí.

Detailní experiment musí poskytovat spolehlivá (nejlépe statisticky významná) data, čemuž je podřízen výběr instancí.

Každý experiment má za úkol odpovědět na nějakou otázku. Musí být navržen tak, aby dal žádanou odpověď, například k testování robustnosti použijeme jednu nebo několik instancí, které permutujeme. Experiment nemá dávat falešné odpovědi, například by neměl odpovídat na otázku omezenou (např. na určitý typ instancí nebo dokonce na jednu velikost). Data tedy musí být reprezentativní, což je vážný problém. Dokonce, pokoušíme-li se data generovat náhodně, můžeme nevědomky generovat instance s určitými pevnými charakteristikami. Proto je generátor instancí rozsáhle parametrizován.

Experimentální vyhodnocení pracuje vždy s konkrétní implementací. Někdy bývá potřebné usoudit na vlastnosti algoritmu; je to obtížné, ale možné. Při diskusi výsledků je třeba oba typy výsledků odlišovat.

Jsou zde jistá podezření na závislosti:

- výpočetní náročnost dynamického programování může být citlivá na maximální váhu nebo cenu,
- výkon metod, které vycházejí ze stavu "prázdný batoh" se může lišit od metod, vycházejících ze stavu "plný batoh" podle poměru celková váha / kapacita batohu,
- není jasné, jakou roli hraje granularita instance (převaha malých nebo převaha velkých věcí).
- je konkrétní implementace heuristiky (zejména metoda větví a hranic) robustní?

O níže diskutovaných závislostech přece jen něco víme. Například, podle použité dekompozice můžeme usoudit, na co není dynamické programování citlivé. Takové vlastnosti ale raději ověřte na úrovni pilotního experimentu. (Inženýrské pořekadlo: myslet znamená málo vědět.)

Pro citlivostní vyhodnocení (pro jiné obecně nikoliv) pravděpodobně stačí zafixovat všechny parametry na konstantní hodnotu a vždy plynule měnit jeden parametr. Neumíme vyloučit, že parametry spolu interagují, ale myslíme si, že ne (viz pořekadlo výše). Je nutné naměřit výsledky pro aspoň čtyři (opravdu minimálně) vhodně zvolené hodnoty parametru, jinak některé závislosti nebude možné vypozorovat.

Zpráva by měla obsahovat aspoň stručný popis jednotlivých algoritmů. Jinak nemusí být jasné, jaký typ B&B, DP, ... byl použit.

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
```

Následně připravíme ještě generátor instancí.

```
cd gen
make all
```

Benchmarking

V této úloze jsem se s měřením výkonu nespoléhal na existující Rust knihovny a namísto toho provedl měření v Pythonu.

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

Srovnání algoritmů

Následující seznam poskytuje přehled implementovaných algoritmů. Jednoduchý hladový přístup ani žádná z variant FPTAS nebyly součástí experimentů.

- `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.

```
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
```

```
<<python-imports>>
```

```
show_progress = os.environ.get("JUPYTER") == None
algs = ["bf", "bb", "dpc", "dpw", "redux"]
data = []

# 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 generate(**kwargs):
    <<generate-instance>>

def solve(alg, instance):
    <<invoke-solver>>

<<dataset-utilities>>


n_samples = 2 # FIXME

<<datasets>>

<<measurement-loop>>

<<performance-chart>>
```

```
# enumerate the parameter values of a dataset for instance generation and
# algorithm benchmarking.
```

```
def dataset(id, **kwargs):
    params = dict({
        # defaults
        "id": [id],
        "alg": algs,
        "seed": [42],
        "n_runs": [1],
        "n_permutations": [1],
        "n_repetitions": [1],
        "n_items": [27],
        "max_weight": [5000],
        "max_cost": [5000],
        "granularity_and_light_heavy_balance": [(1, "bal")],
        "capacity_weight_sum_ratio": [0.8],
        "cost_weight_correlation": ["uni"],
    }, **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]
    }
# benchmark configurations
# we don't want a full cartesian product (too slow to fully explore), so we'll
# use a union of subsets, each tailored to the particular algorithm
configs = merge_datasets(dataset(
    "weight range",
    alg = ["bf", "dpw"],
    max_weight = [500, 1000, 5000, 10000, 50000, 100000, 500000, 1000000],
), dataset(
    "cost range",
    alg = ["bf", "dpc"],
    max_cost = [500, 1000, 5000, 10000, 50000, 100000, 500000],
), dataset(
    "n_items range",
    n_items = [4, 10, 15, 20, 25, 28],
), dataset(
    "granularity exploration",
    alg = ["bb", "dpc", "dpw", "redux"],
    n_runs = [n_samples],
    granularity_and_light_heavy_balance = [
        (1, "light"), (2, "light"), (3, "light"), (1, "heavy"), (2, "heavy"), (3, "heavy")
```

```
    ],
), dataset(
    "capacity weight sum ratio exploration",
    alg = ["bb", "dpc", "dpw", "redux"],
    n_runs = [n_samples],
    capacity_weight_sum_ratio = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9],
), dataset(
    "branch and bound robustness",
    seed = [420],
    n_items = [18], # FIXME
    alg = ["bf", "bb", "dpw", "redux"],
    n_permutations = [20],
    n_repetitions = [10],
))

res = []
kwargs["granularity"] = kwargs["granularity_and_light_heavy_balance"][0]
kwargs["light_heavy_balance"] = kwargs["granularity_and_light_heavy_balance"][1]
del kwargs["granularity_and_light_heavy_balance"]
for seed in range(kwargs["seed"], kwargs["seed"] + kwargs["n_runs"]):
    params = dict({
        "seed": seed,
        "n_instances": 1,
    }, **kwargs)
    # run the instance generator
    instance = dict({"contents": os.popen(
        "gen/kg2 \
        -r {seed} \
        -n {n_items} \
        -N {n_instances} \
        -W {max_weight} \
        -C {max_cost} \
        -k {granularity} \
        -w {light_heavy_balance} \
        -c {cost_weight_correlation} \
        -m {capacity_weight_sum_ratio} \
        ".format(**params)
    ).read()}, **params)

    for p in range(0, instance["n_permutations"]):
        kg_perm = run(
            "gen/kg_perm \
            -d 0 \
            -N 1 \
            -r {} \
            ".format(p).split(),
            stdout = PIPE,
            stderr = PIPE,
            input = instance["contents"],
            encoding = "ascii",
        )
```

```
        res.append(dict({
            "contents": kg_perm.stdout,
            "perm_id": p,
        }, **instance))

return res
```

```
solver = run(
    ["target/release/main", alg],
    stdout = PIPE,
    stderr = PIPE,
    input = instance["contents"],
    encoding = "ascii",
    cwd = "solver/"
)
if solver.returncode != 0:
    print(solver)
    raise Exception("solver failed")

# return only the cost of the solution
return int(solver.stdout.split()[0])
```

```
iteration = 0
total = sum([r * p * rep for (r, p, rep) in zip(configs["n_runs"], configs["n_permutations"], configs
for config in [dict(zip(configs, v)) for v in zip(*configs.values())]:
    param_iter = iter(config.values())
    next(param_iter) # skip id
    if show_progress:
        print(end = "\033[2K") # clear the current line to get rid of the progress bar
    print(config["id"], "\tparams", *param_iter)
    progress_bar(iteration, total)

    for inst in generate(**config):
        for rep in range(0, config["n_repetitions"]):
            # measure the time taken by the call to the solver
            start = time.time()
            cost = solve(config["alg"], inst)
            end = time.time()
            data.append(dict(inst,
                cost = cost,
                alg = config["alg"],
                t = end - start,
                repetition = rep,
                contents = None
            ))
            iteration = iteration + 1
            progress_bar(iteration, total)

print()
```

```
# plot the measurements

figsize = (14, 8)
```

```
<<chart-labels>>

def plot(x_axis, y_axis, id, title, data = data, filename = None):
    if filename is None:
        filename = id.replace(" ", "_")
    print("\t{}".format(title))
    fig, ax = plt.subplots(figsize = figsize)
    ds = [d for d in data if d["id"] == id]
    # create a frame from the list
    df = pd.DataFrame(ds)

    # do a boxplot grouped by the algorithm name
    <<plot-boxplot>>

    # render the datapoints as dots with horizontal jitter
    <<plot-stripplot>>

    plt.title(title)
    plt.xlabel(plot_labels[x_axis])
    plt.ylabel(plot_labels[y_axis])

    <<plot-caption>>

    handles, labels = ax.get_legend_handles_labels()
    labels = [alg_labels[l] for l in labels]

    plt.legend(handles[0 : int(len(handles) / 2)], labels[0 : int(len(labels) / 2)])
    plt.savefig("docs/assets/{}.svg".format(filename))

print("rendering plots")
<<plots>>
```

```
sns.boxplot(
    x = x_axis,
    y = y_axis,
    data = df,
    hue = "alg",
    ax = ax,
    linewidth = 0.8,
)
```

```
sns.stripplot(
    x = x_axis,
    y = y_axis,
    data = df,
    hue = "alg",
    ax = ax,
    jitter = True,
    size = 4,
    dodge = True,
    linewidth = 0.2,
    alpha = 0.4,
```

```
    edgecolor = "white",
)

constant_columns = [
    col for col in df.columns[df.nunique() <= 1]
        if (col not in ["id", "n_instances", "contents"])
]

caption = "\n".join(tr.wrap("Konfigurace: {}".format({
    k: df[k][0] for k in constant_columns
}), width = 170))

fig.text(
    0.09,
    0.05,
    caption,
    fontsize = "small",
    fontfamily = "monospace",
    verticalalignment = "top",
)

plot("n_items",     "t", "n_items range",           "Průměrná doba běhu vzhledem k velikosti instance
plot("max_weight",  "t", "weight range",            "Průměrná doba běhu vzhledem k maximální váze")
plot("max_cost",    "t", "cost range",              "Průměrná doba běhu vzhledem k maximální ceně")


for balance in ["light", "heavy"]:
    plot(
        "granularity",
        "t",
        "granularity exploration",
        "Doba běhu vzhledem ke granularitě (preference {})".format(balance),
        data = [d for d in data if d["light_heavy_balance"] == balance],
        filename = "granularity_exploration_{}".format(balance),
    )

plot(
    "capacity_weight_sum_ratio",
    "t",
    "capacity weight sum ratio exploration",
    "Doba běhu vzhledem k poměru kapacity a součtu vah",
)

plot(
    "perm_id",
    "t",
    "branch and bound robustness",
    "Doba běhu přes několik permutací jedné instance",
)

plot(
    "perm_id",
    "cost",
```

```
    "branch and bound robustness",
    "Cena řešení přes několik permutací jedné instance",
    filename = "branch_and_bound_robustness_cost"
)

plot_labels = dict(
    seed = "Seed",
    t = "Doba běhu [s]",
    cost = "Cena řešení",
    perm_id = "ID permutace",
    n_items = "Velikost instance",
    max_cost = "Maximální cena",
    max_weight = "Maximální váha",
    n_instances = "Počet instancí v zadání",
    granularity = "Granularita",
    light_heavy_balance = "Rozložení váhy předmětů",
    capacity_weight_sum_ratio = "Poměr kapacity a součtu vah",
)

alg_labels = dict(
    bf = "Brute force",
    bb = "Branch & bound",
    dpc = "Dynamic programming (cost)",
    dpw = "Dynamic programming (weight)",
    redux = "Greedy redux",
)
```

Výkon každého algoritmu je na instancích měřen jednotlivě, tj. generátor instancí je vždy instruován k výpisu jediné instance, která je následně permutována jak je potřeba a nakonec předána příslušnému řešiči.

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

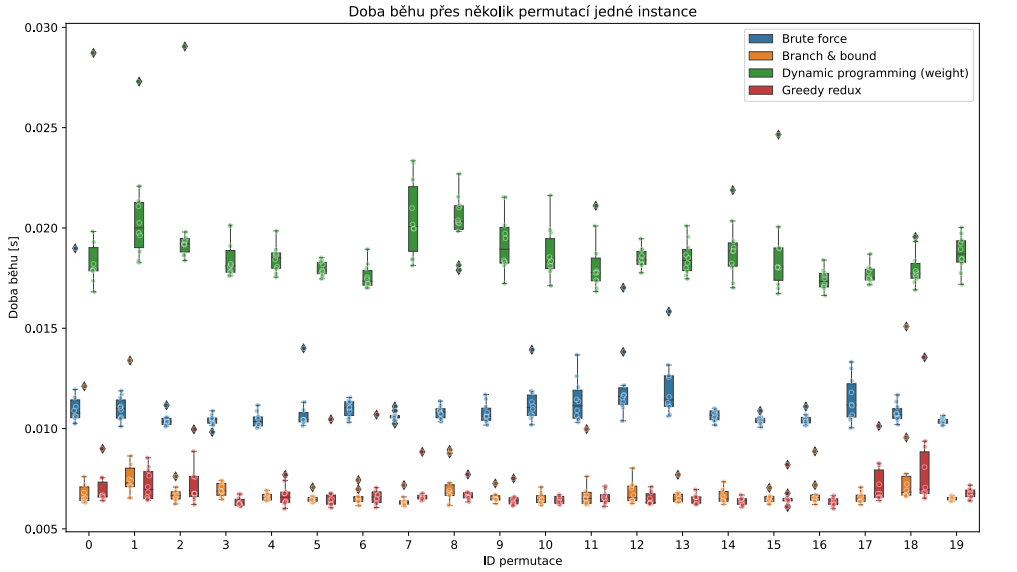


Figure 1: branch and bound robustness

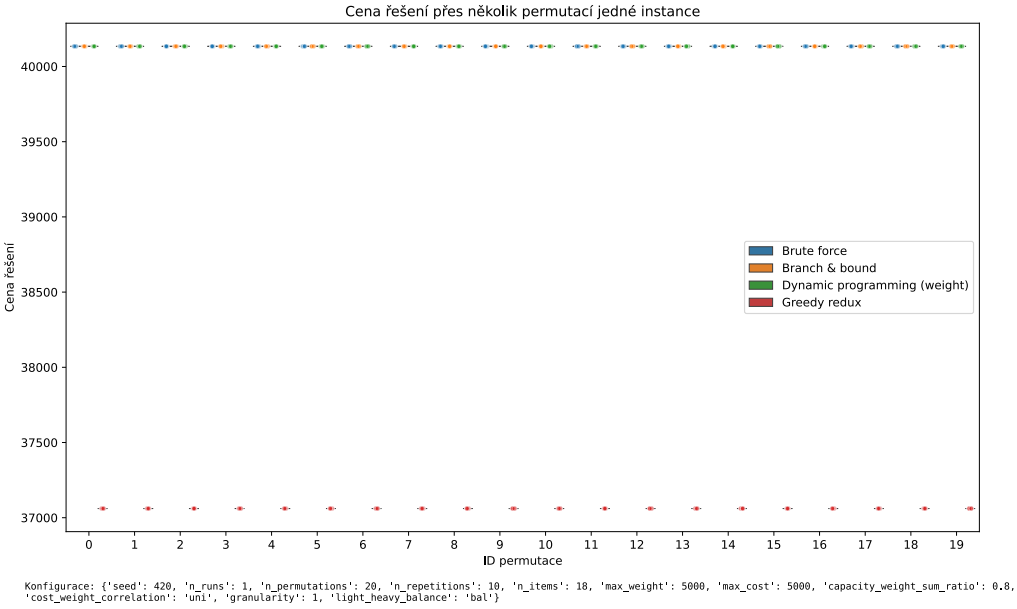
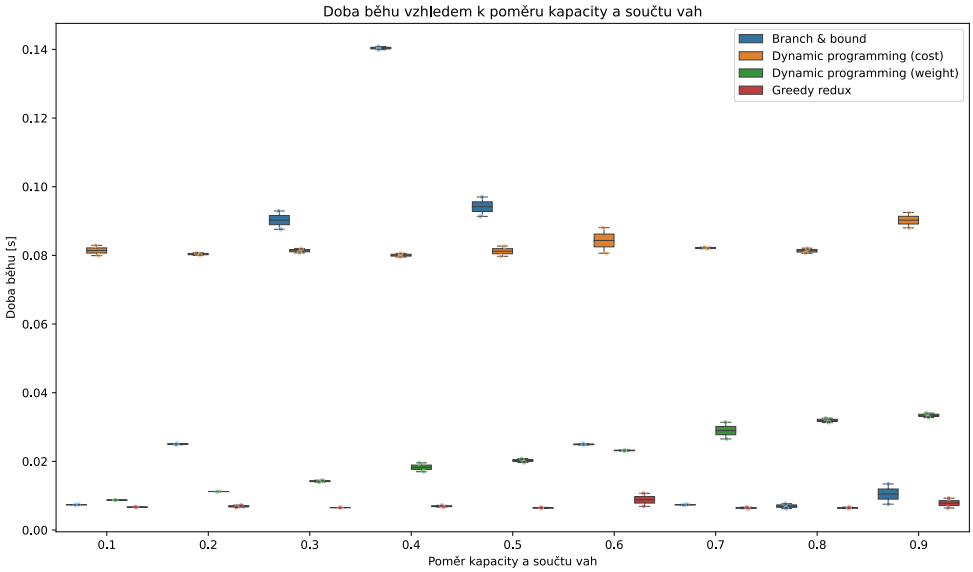
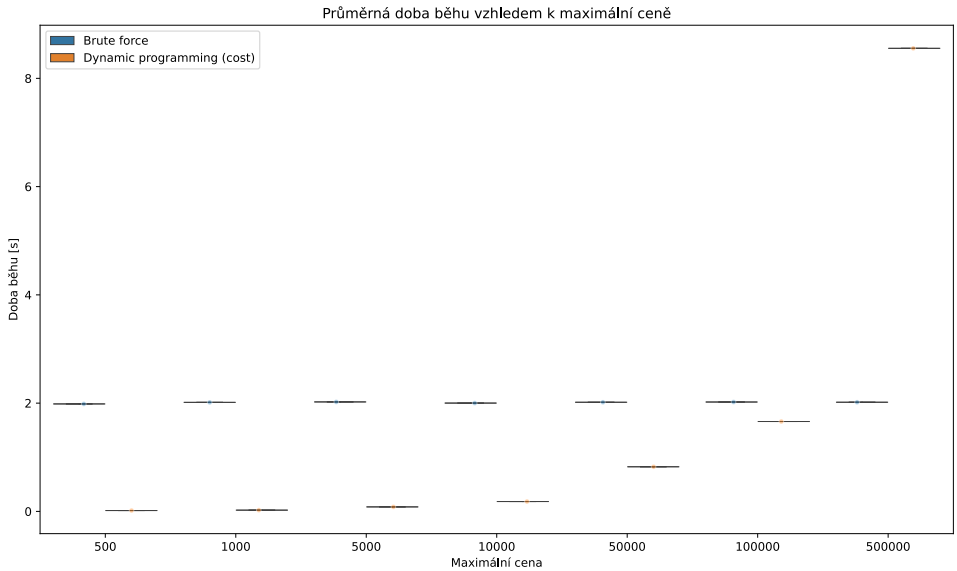


Figure 2: branch and bound robustness - cost



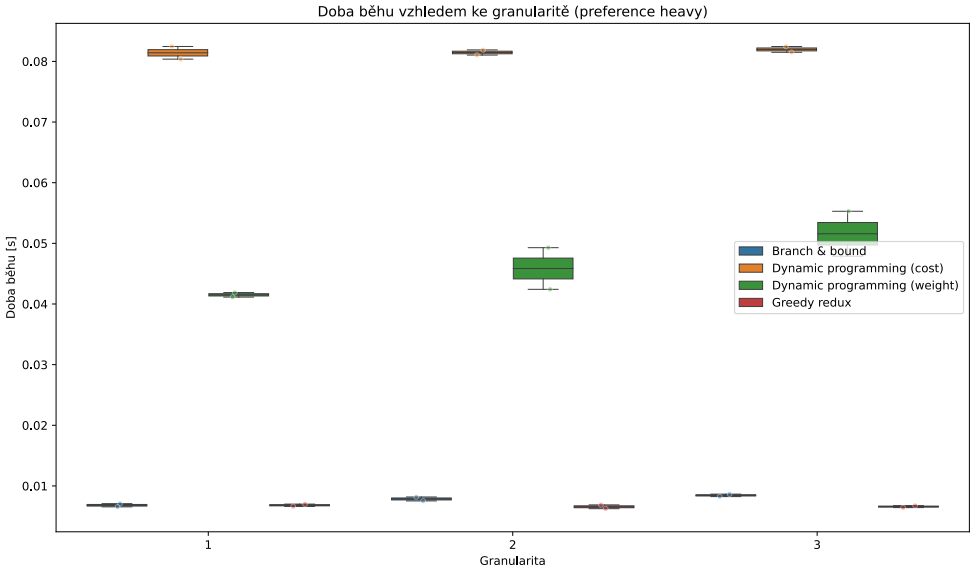
Konfigurace: {'perm_id': 0, 'seed': 42, 'n_runs': 2, 'n_permutations': 1, 'n_repetitions': 1, 'n_items': 27, 'max_weight': 5000, 'max_cost': 5000, 'cost_weight_correlation': 'uni', 'granularity': 1, 'light_heavy_balance': 'bal', 'repetition': 0}

Figure 3: capacity weight sum ratio exploration



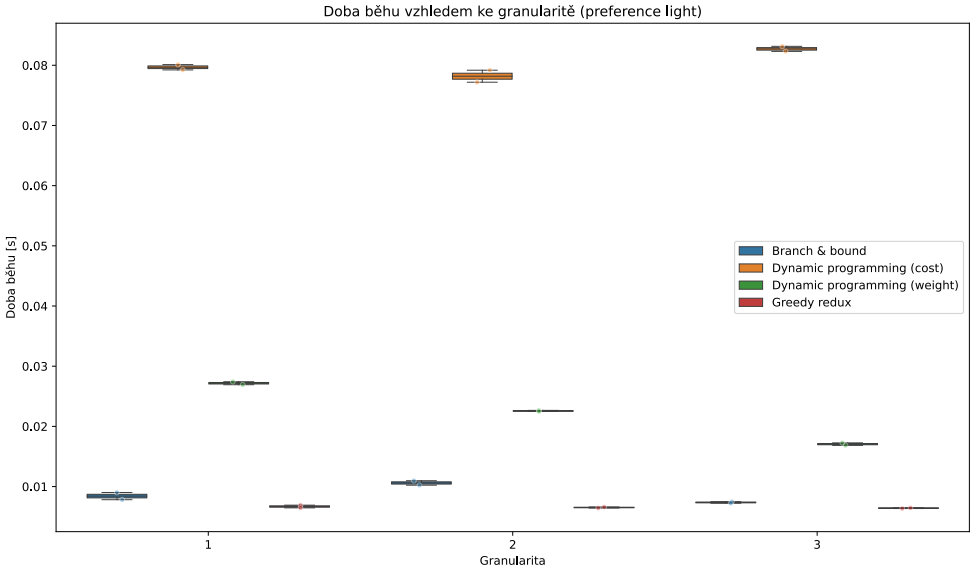
Konfigurace: {'perm_id': 0, 'seed': 42, 'n_runs': 1, 'n_permutations': 1, 'n_repetitions': 1, 'n_items': 27, 'max_weight': 5000, 'capacity_weight_sum_ratio': 0.8, 'cost_weight_correlation': 'uni', 'granularity': 1, 'light_heavy_balance': 'bal', 'repetition': 0}

Figure 4: cost range



Konfigurace: {'perm_id': 0, 'seed': 42, 'n_runs': 2, 'n_permutations': 1, 'n_repetitions': 1, 'n_items': 27, 'max_weight': 5000, 'max_cost': 5000, 'capacity_weight_sum_ratio': 0.8, 'cost_weight_correlation': 'uni', 'light_heavy_balance': 'heavy', 'repetition': 0}

Figure 5: granularity exploration heavy



Konfigurace: {'perm_id': 0, 'seed': 42, 'n_runs': 2, 'n_permutations': 1, 'n_repetitions': 1, 'n_items': 27, 'max_weight': 5000, 'max_cost': 5000, 'capacity_weight_sum_ratio': 0.8, 'cost_weight_correlation': 'uni', 'light_heavy_balance': 'light', 'repetition': 0}

Figure 6: granularity exploration light

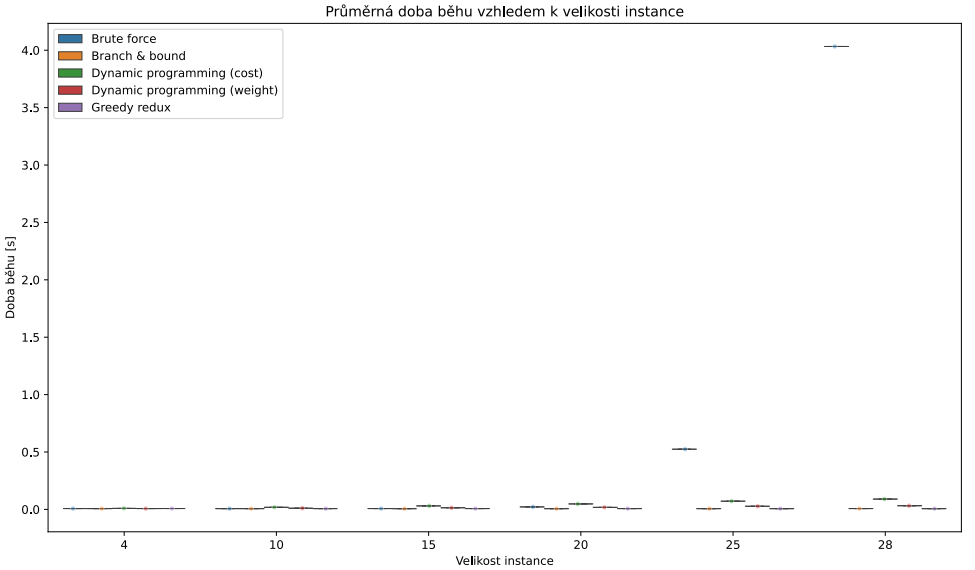


Figure 7: n_items range

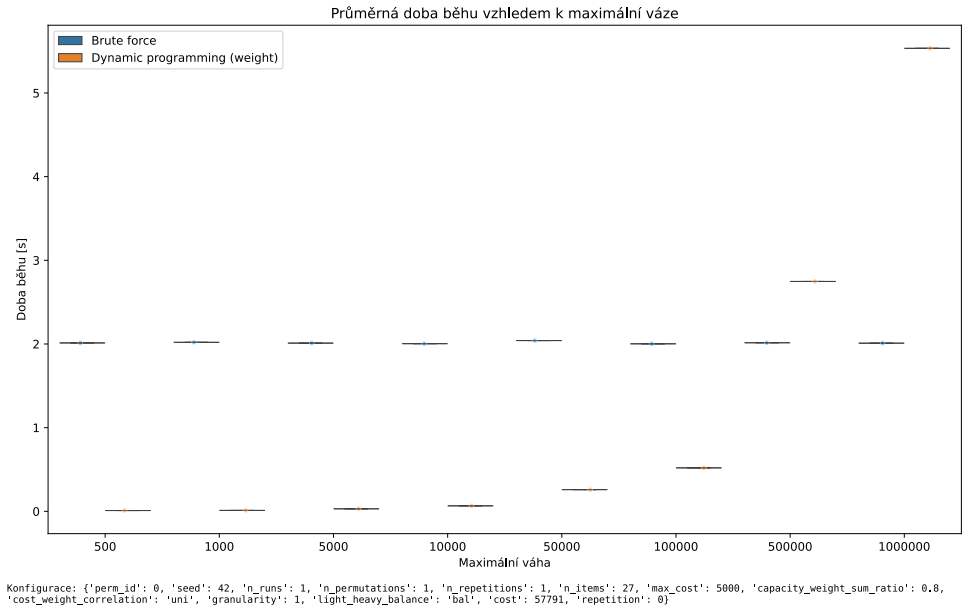


Figure 8: weight range

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, 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)
        }
    }
}
```

```rust
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.

```rust
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 }
```

```rust
#[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.

```rust
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 `BTreeMapě`. Tu používáme při vybírání algoritmu pomocí argumentu předaného na příkazové řádce, v testovacím kódu na testy všech implementací atp.

```rust
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

```
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).

```rust
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.

```rust
// 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

TODO

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

```rust
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.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(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};
    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);
    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(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();
}
```

```rust
    #[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();
    }
}
```