NI-KOP – úkol 4

Ondřej Kvapil

Kombinatorická optimalizace: problém batohu

Zadání

Účel

V této úloze si budete moci vyzkoušet nasazení pokročilé iterativní techniky "v malém", na řešení jednoduchého problému (problém batohu). Proto by jak seznámení se s heuristikou samotnou, tak white box a black box fáze měly být snadné. Techniky experimentálního vyhodnocení, které jste si osvojili v předchozích úlohách, zde použijete k ověření, zda Vaše nasazení heuristiky je dostatečně kvalitní. V poslední úloze (jejíž hodnocení bude součástí hodnocení zkoušky) budete řešit složitější problém touto pokročilou technikou, tudíž "osahání si" těchto algoritmů a jejich nasazení na jednodušší úloze bude pro Vás přínosem. Krok od 4. úlohy k 5. bude pak malý: pouze upravíte konfiguraci a optimalizační kritérium pro jiný typ problému. Seznámíte se s reakcí algoritmu na manipulaci s parametry.

Problém 5. úlohy má ovšem podstatně jiné vlastnosti, než problém batohu. Nepočítejte s tím, že kromě kódu přeberete i konkrétní hodnoty parametrů.

Algoritmy k výběru

- Simulované ochlazování
- Genetické algoritmy
- Tabu search

Úkol

- Zvolte si heuristiku, kterou budete řešit problém vážené splnitelnosti booleovské formule (simulované ochlazování, simulovaná evoluce, tabu prohledávání)
- Tuto heuristiku použijte pro řešení problému batohu. Můžete použít dostupné instance problému (instance zde, popis zde), anebo si vygenerujte své instance pomocí generátoru. Používejte instance s větším počtem věcí (>30).
- Hlavním cílem domácí práce je seznámit se s danou heuristikou, zejména se způsobem, jakým se nastavují její parametry (rozvrh ochlazování, selekční tlak, tabu lhůta…) a modifikace (zjištění počáteční teploty, mechanismus selekce, tabu atributy…). Není-li Vám cokoli jasné, prosíme, ptejte se na cvičeních.
- Proveďte obě fáze nasazení (white box, black box) a podejte o nich zprávu.

Pokyny

Problém batohu není příliš obtížný, většinou budete mít k dispozici globální maxima (exaktní řešení) z předchozích prací, například z dynamického programování. Při správném nastavení parametrů byste měli vždy dosáhnout těchto optim, případně pouze velice malých chyb. Doba výpočtu může ovšem být relativně větší. Závěrečná úloha je co do nastavení a požadovaného výkonu heuristiky podstatně náročnější a může vyžadovat zcela jiné nastavení parametrů.

Zpráva by měla dokumentovat, že jste si při nasazení počínali racionálně. Obsahovat by měla:

- Stručný popis zvoleného algoritmu.
- Popis white box fáze. Postup nastavení parametrů, výsledky experimentů s různými nastaveními parametrů, které vedly k určitému rozhodnutí, typicky vývoj řešení (populace u GA) v průběhu běhu algoritmu. Graf může dobře ilustrovat závěr, který z dat děláte.
- Popis black box fáze. Výsledek měření = čas výpočtu a rel. chyba. Pokud nejste schopni vypočítat rel. chybu, stačí uvést vývoj výsledné ceny (počáteční → koncová)
- Pokuste se vyvodit nějaké závěry

Poznámka

- Hodnocení této úlohy bude odpovídat účelu úlohy - tj. prozkoumat chování příslušné pokročilé heuristiky. Hodnocení tedy bude velice benevolentní co se dosažených výsledků týče, méně už v případě postupu nasazení.

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

```
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 generování vstupních instancí, spouštění řešiče a měření času, který problém řešiči zabere. Při spuštění v terminálu skript zároveň hlásí průběžný postup měření.

```
<<python-imports>>

<<performance-chart>>
```

Jelikož by průchod hrubou silou přes všechny možné kombinace parametrů generátoru zabral zbytečně dlouho, definujeme prostor parametrů jako sjednocení zajímavých podprostorů. Tyto podprostory nazýváme "datasety" – v tomto programu je reprezentují slovníky, které jednotlivým parametrům generátoru, řešiče a podpůrné infrastruktury přiřazují seznamy možných hodnot. Každý dataset má navíc unikátní klíč, podle kterého jej lze identifikovat při vizualizaci.

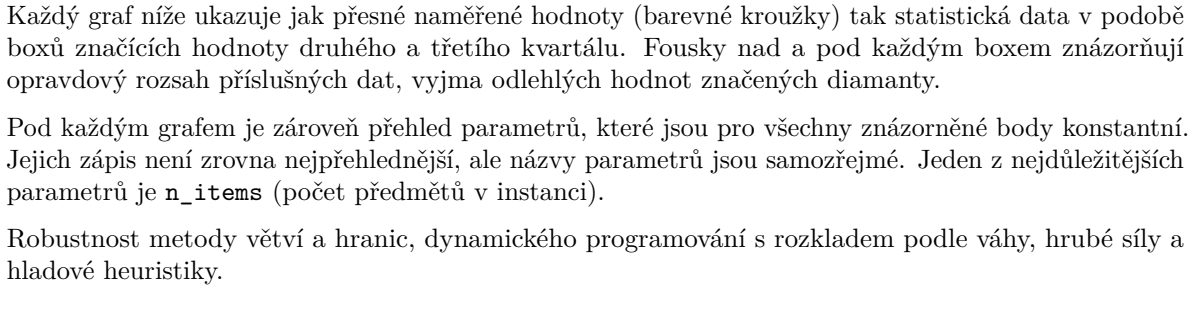
Hlavní smyčka pro měření jednoduše projde všechny možné konfigurace dané sjednocením datasetů, vygeneruje odpovídající instance a spustí na ně příslušný algoritmus. Parametr `n_repetitions` určuje kolikrát změřit jednu konfiguraci (stejná instance, stejná permutace, stejný algoritmus) – měření provádíme vícekrát, abychom odhalili chyby měření.

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.

```
# plot the measurements

figsize = (14, 8)

def invoke_solver(input, cfg):
    solver = run(
        ["target/release/main", "sa"] + list(cfg),
        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)

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

errors = []
for cfg in product(
    # max iterations
    # ["15000"],
    ["3000"],
    # scaling factor
    ["0.85", "0.9", "0.95", "0.99", "0.991", "0.992", "0.993", "0.994", "0.995", "0.997"],
    # temperature modifier
    ["1"],
):
    print(cfg)
    params = '-'.join(cfg)
    for instance in input.split("\n")[:10]:
        id = instance.split()[0]
        (t, cost, err, cost_temperature_progression) = invoke_solver(instance, cfg)
        errors.append({"scaling factor": cfg[1], "error": err})
        print("took", t, "ms")

        # 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()

print(*errors, sep = "\n")

df = pd.DataFrame(errors)
series = df.groupby("scaling factor")["error"].mean()
df["mean error"] = df["scaling factor"].map(series)

# plot the error distributions for each scaling factor
```

```
plt.style.use("default")
sns.set_theme(style = "white", rc = {"axes.facecolor": (0, 0, 0, 0)})
pal = sns.color_palette("crest", n_colors = len(set([e["scaling factor"] for e in errors])))

# set up the layout
g = sns.FacetGrid(
    df,
    row = "scaling factor",
    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, alpha =
# contours
g.map(sns.kdeplot, "error", clip = (-0.1, 1.0), bw_adjust = 1, clip_on = False, color = "w", lw = 1)
# 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, 1, df["scaling factor"].unique()[i],
            fontsize = 15, color = ax.lines[-1].get_color())

# remove titles, y ticks, spines
g.set_titles("")
g.set(yticks = [])
g.despine(left = True, bottom = True)
g.fig.suptitle("Vliv koeficientu chlazení na hustotu chyb", 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("Chyba oproti optimálnímu řešení [%]")
g.set_ylabels("")

g.savefig("docs/assets/whitebox-error-distributions.svg")

# sns.kdeplot([100 * e for e in errors], shade = True)
# plt.savefig(f"docs/assets/whitebox-overview-{params}.svg")
# plt.close()
```



Každý graf níže ukazuje jak přesné naměřené hodnoty (barevné kroužky) tak statistická data v podobě boxů značících hodnoty druhého a třetího kvartálu. Fousky nad a pod každým boxem znázorňují opravdový rozsah příslušných dat, vyjma odlehlých hodnot značených diamanty.

Pod každým grafem je zároveň přehled parametrů, které jsou pro všechny znázorněné body konstantní. Jejich zápis není zrovna nejpřehlednější, ale názvy parametrů jsou samozřejmé. Jeden z nejdůležitějších parametrů je `n_items` (počet předmětů v instanci).

Robustnost metody větví a hranic, dynamického programování s rozkladem podle váhy, hrubé síly a hladové heuristiky.

Robustnost výše uvedených algoritmů podle ceny řešení. Žádný z měřených algoritmů nevykazuje známky závislosti na zápisu instance.

Závislost doby běhu na poměru kapacity a součtu vah předmětů. Nelineárně se prokazuje u metody větví a hranic, lineární závislost je zřejmá u dynamického programování s rozkladem podle váhy.

Závislost doby běhu na maximální ceně předmětů. Očekávané zhoršení výkonu dynamického programování s rozkladem podle ceny je jasně vidět.

Závislost doby běhu na maximální váze předmětů. Očekávané zhoršení výkonu je znovu vidět u dynamického programování, tentokrát s rozkladem podle váhy.

Závislost doby běhu na granularitě s preferencí těžších předmětů.

Závislost doby běhu na granularitě s preferencí lehčích předmětů.

Závislost doby běhu na počtu předmětů v instanci. Tento vztah už důvěrně známe z předchozích úkolů.

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>>

pub fn simulated_annealing<Rng>(
    &self,
    rng: &mut Rng,
    (max_iterations, scaling_factor, temp_modifier): (u32, f64, f64)
) -> 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 % (self.items.len() as u32 / 4) == 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 {} a
```

```
                                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 th
                           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
    }
}

<<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.

```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> {
        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

```
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.

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

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

        for cap in 0 ..= *m as usize {
            let s = if (cap as u32) < weight {
                    last[cap]
```

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

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

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 }
}
```

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.

```
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};
```

```rust
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:

```rust
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()?;
        Ok(Left((max_iterations, scaling_factor, temp_modifier)))
    } } 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.

```rust
#[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())
        }
    }

    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(())
    }
```

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