NI-KOP – úkol 1

Ondřej Kvapil

Kombinatorická optimalizace: problém batohu

Zadání

- Experimentálně vyhodnoťte závislost výpočetní složitosti na velikosti instance u následujících algoritmů rozhodovací verze 0/1 problému batohu:
  - hrubá síla
  - metoda větví a hranic (B&B)
- Otázky, které má experiment zodpovědět:
  - Vyhovují nejhorší případy očekávané závislosti?
  - Závisí střední hodnota výpočetní závislosti na sadě instancí? Jestliže ano, proč?

Pokyny

Oba algoritmy naprogramujte. Výpočetní složitost (čas) je nejspolehlivější a nejjednodušší měřit počtem navštívených konfigurací, to jest vyhodnocených sestav věcí v batohu. Na obou sadách pozorujte závislost výpočetního času na n, pro n v rozsahu, jaký je Vaše výpočetní platforma schopna zvládnout, a to jak maximální, tak průměrný čas. Pro alespoň jednu hodnotu n (volte instance velikosti alespoň 10) zjistěte četnosti jednotlivých hodnot (histogram) a pokuste se jej vysvětlit. Ohledně metody větví a hranic – uvědomte si, že se jedná o rozhodovací problém a podle toho ořezávejte. Nápověda: i když je to rozhodovací problém, lze použít ořezávání podle ceny. Jak? Implementované způsoby ořezávání popište ve zprávě.

Sady NR a ZR vyhodnocujte zvlášť a proveďte jejich srovnání (stačí diskuze).

Bonusový bod

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

- Zjištění, jak čas CPU souvisí s počtem vyhodnocených konfigurací na Vaší platformě a jak je tato závislost stabilní při opakovaném měření téže instance.
- Nový (a experimentálně porovnaný) způsob prořezávání v metodě větví a hranic.
- atd.

Řešení

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

Instrukce k sestavení programu

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

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

Benchmarking

Pro provedení měření výkonu programu jsem využil nástroje Hyperfine.

```
uname -a
./cpufetch --logo-short --color ibm
mkdir -p docs/measurements/
cd solver
hyperfine --export-json ../docs/bench.json \
          --parameter-list n 4,10,15,20,22,25 \
          --parameter-list set N,Z \
          --parameter-list alg bf,bb \
          --min-runs 4 \
          --style color \
          'cargo run --release -- {alg} \
           < ds/{set}R{n}_inst.dat \
           > ../docs/measurements/{alg}-{set}{n}.txt' 2>&1 \
    | fold -w 120 -s
```

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

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

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

alg.	sada	n	průměr	$\pm\sigma$	minimum	medián	maximum
bf	N	4	**99**.28 ms	**3**.94 ms	**96**.52 ms	**98**.32 ms	**118**.86 ms
bf	N	10	**103**.52 ms	**1**.24 ms	**100**.87 ms	**103**.59 ms	**107**.41 ms
bf	N	15	**225**.66 ms	**4**.09 ms	**221**.03 ms	**225**.58 ms	**231**.12 ms
bf	N	20	**4275**.64 ms	**74**.03 ms	**4214**.95 ms	**4258**.38 ms	**4370**.86 ms
bf	N	22	**15292**.73 ms	**482**.8 ms	**14868**.94 ms	**15286**.04 ms	**15729**.92 ms
bf	N	25	**124713**.12 ms	**12**.92 ms	**124694**.59 ms	**124716**.71 ms	**124724**.46 ms

alg.	sada	n	průměr	$\pm\sigma$	minimum	medián	maximum
bf	Z	4	**100**.27 ms	**1**.55 ms	**97**.12 ms	**100**.37 ms	**102**.71 ms
bf	Z	10	**107**.3 ms	**1**.61 ms	**104**.78 ms	**106**.79 ms	**110**.81 ms
bf	Z	15	**296**.07 ms	**6**.64 ms	**289**.88 ms	**293**.82 ms	**310**.35 ms
bf	Z	20	**6613**.03 ms	**235**.71 ms	**6392**.77 ms	**6620**.85 ms	**6817**.64 ms
bf	Z	22	**25812**.3 ms	**2**.34 ms	**25810**.36 ms	**25811**.65 ms	**25815**.55 ms
bf	Z	25	**205518**.94 ms	**1794** ms	**204498**.54 ms	**204689**.88 ms	**208197**.46 ms
bb	N	4	**103**.68 ms	**1**.47 ms	**100**.19 ms	**103**.82 ms	**105**.96 ms
bb	N	10	**103**.91 ms	**1**.86 ms	**100**.95 ms	**104**.06 ms	**107**.77 ms
bb	N	15	**118**.26 ms	**1**.32 ms	**115**.42 ms	**118**.44 ms	**120**.94 ms
bb	N	20	**446**.94 ms	**12**.66 ms	**439**.56 ms	**442**.58 ms	**472**.64 ms
bb	N	22	**1473**.91 ms	**38**.15 ms	**1441**.72 ms	**1468**.31 ms	**1517**.29 ms
bb	N	25	**8652**.92 ms	**2**.19 ms	**8650**.57 ms	**8652**.68 ms	**8655**.75 ms
bb	Z	4	**103**.09 ms	**1**.38 ms	**100**.68 ms	**103**.09 ms	**106**.14 ms
bb	Z	10	**107**.22 ms	**2**.16 ms	**104**.17 ms	**107**.06 ms	**111**.55 ms
bb	Z	15	**173**.9 ms	**1**.33 ms	**171**.85 ms	**173**.81 ms	**176**.73 ms
bb	Z	20	**1834**.74 ms	**0**.63 ms	**1834**.01 ms	**1834**.77 ms	**1835**.42 ms
bb	Z	22	**6060**.27 ms	**1**.93 ms	**6058**.13 ms	**6060**.19 ms	**6062**.57 ms
bb	Z	25	**42120**.19 ms	**7**.28 ms	**42110**.15 ms	**42122**.38 ms	**42125**.85 ms

Srovnání algoritmů

```
<<preprocessing>>

<<performance-chart>>

<<histogram>>
```

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

bench = pd.read_csv("docs/bench.csv", dtype = "string")
bench.rename({
        "alg.": "alg",
        "$n$": "n",
        "sada": "set",
        "průměr": "avg",
        "$\pm \sigma$": "sigma",
        "medián": "median",
        "minimum": "min",
        "maximum": "max",
    },
    inplace = True,
    errors  = "raise",
    axis    = 1,
)

numeric_columns = ["n", "avg", "sigma", "min", "median", "max"]
bench[numeric_columns] = bench[numeric_columns].apply(lambda c:
```

```
    c.apply(lambda x:
        to_numeric(x.replace("**", "").replace(" ms", ""))
    )
)

# Create a figure and a set of subplots.
fig, ax = plt.subplots(figsize = (11, 6))
labels = { "bf": "Hrubá síla"
         , "bb": "Branch & bound"
         , "dp": "Dynamické programování"
         }

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

# Axis metadata: ticks, scaling, margins, and the legend
plt.xticks(bench["n"])
ax.set_yscale("log", base = 10)
ax.set_yticks(list(plt.yticks()[0]) + list(bench["avg"]), minor = True)
ax.margins(0.05, 0.1)
ax.legend(loc="upper left")

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

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

```
import numpy as np
import os

# Load the data
data = []

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

df = pd.DataFrame(data)

# Plot the histograms

for alg in df.alg.unique():
    plt.figure()
    plt.xlabel('Počet konfigurací')
    plt.ylabel('Četnost výskytu')
```

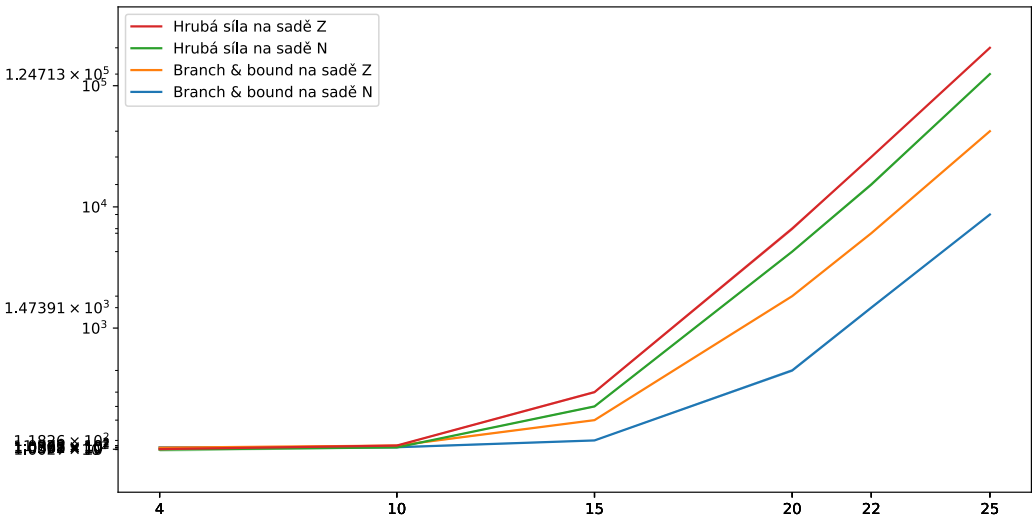

Figure 1: Závislost doby běhu na počtu předmětů. Částečně průhledná oblast značí směrodatnou odchylku (σ).

```
    plt.hist(df[df.alg == alg].n, color = 'tab:blue' if alg[-3] == 'N' else 'orange', bins = 20)#bins
    plt.xlim(xmin = 0)
    # plt.gca().set_xscale('log')
    plt.savefig('docs/assets/histogram-' + alg + '.svg')
    plt.close()
```

Analýza

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

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

Implementace

Program začíná definicí datové struktury reprezentující instanci problému batohu.

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

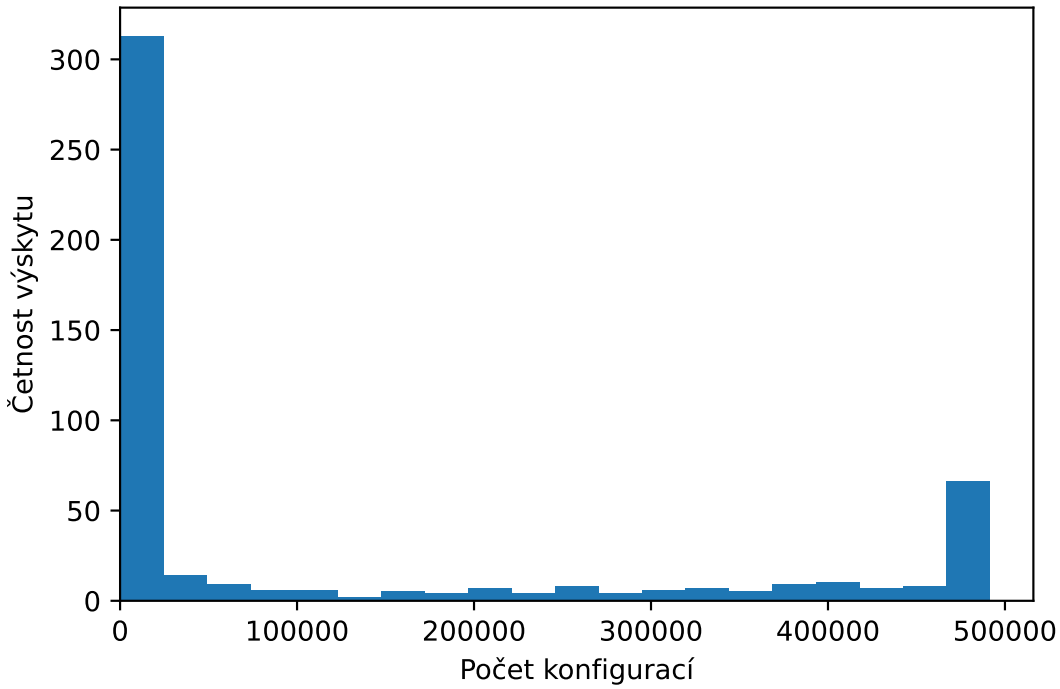

Figure 2: Sada NR: Hrubá síla pro $n = 15$

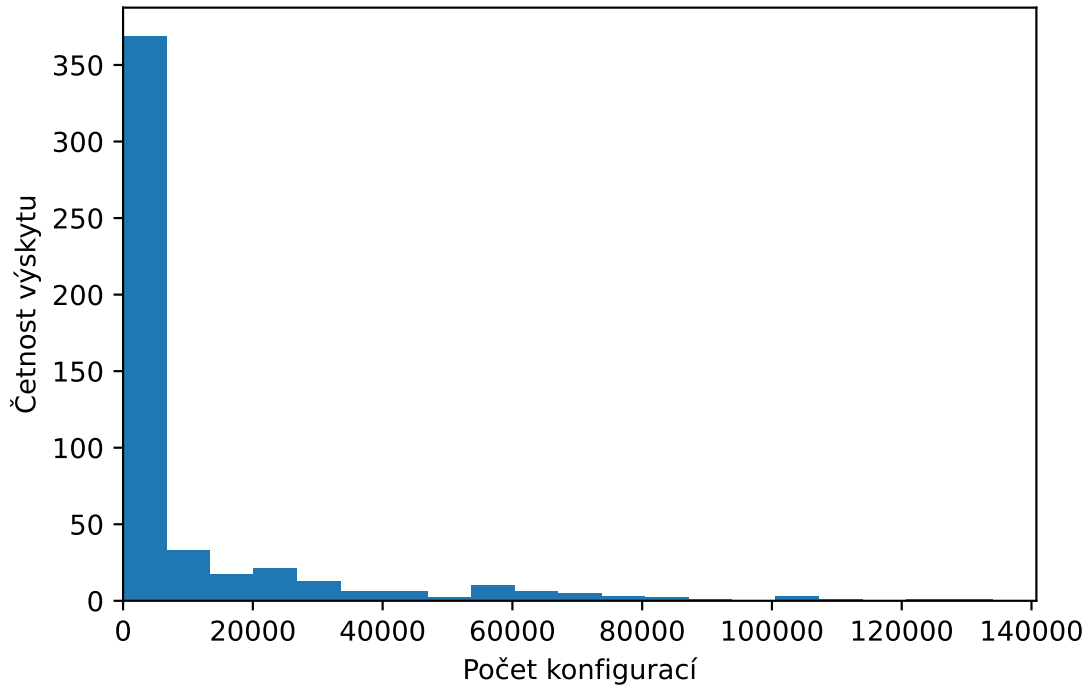

Figure 3: Sada NR: Metoda větví a hranic pro $n = 15$

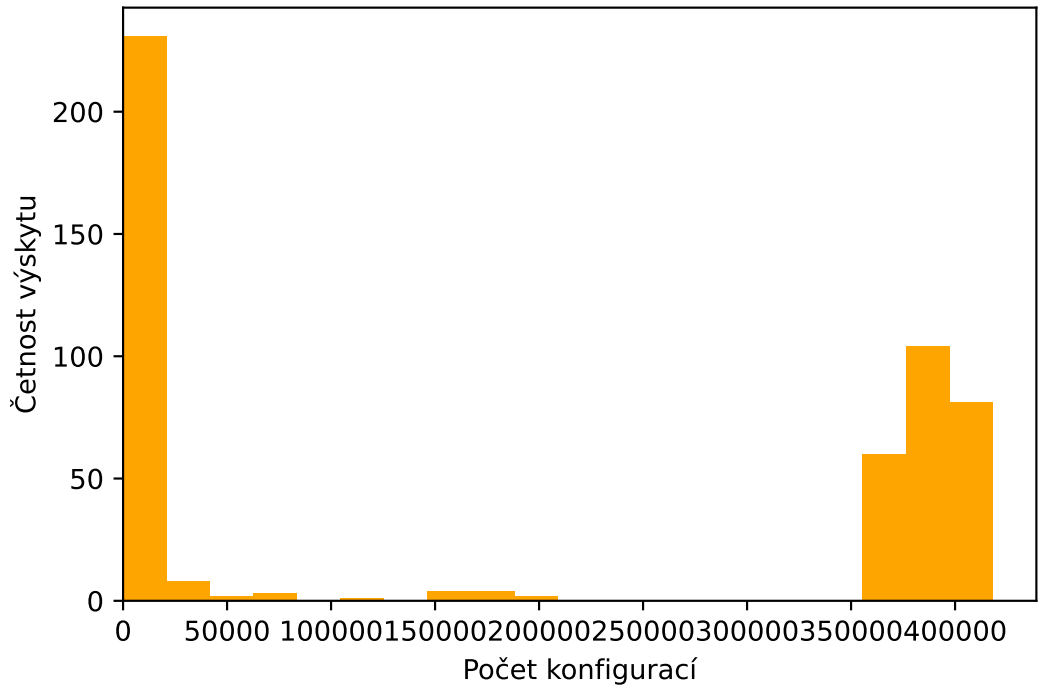

Figure 4: Sada ZR: Hrubá síla pro $n = 15$

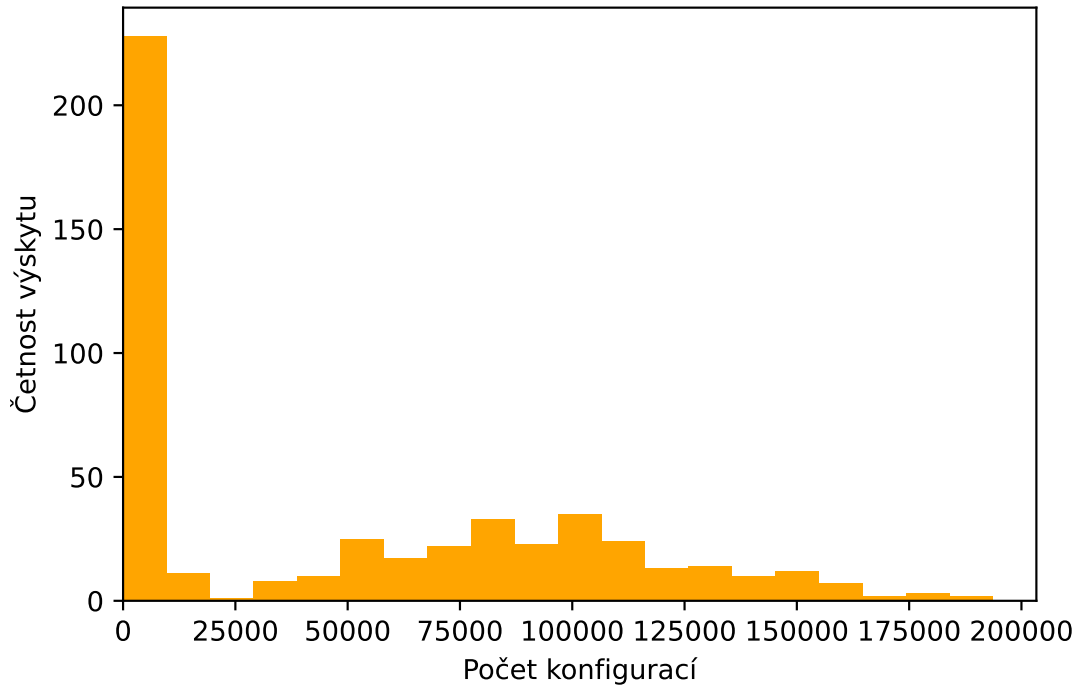

Figure 5: Sada ZR: Metoda větví a hranic pro $n = 15$

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>>

<<problem-instance-definition>>

fn main() -> Result<()> {
    let alg = {
        <<select-algorithm>>
    }?;

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

<<parser>>

<<solution-definition>>

impl Instance {
    <<solver-dp>>

    <<solver-bb>>

    <<solver-bf>>
}

<<tests>>
```

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

```
type Config = BitArr!(for 64);
#[derive(PartialEq, Eq, Clone, Copy, Debug)]
struct Solution<'a> { weight: u32, cost: u32, cfg: Config, visited: u64, inst: &'a Instance }

<<solution-helpers>>
```

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

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

```
        Some(match cost.cmp(&other.cost) {
            Ordering::Equal => weight.cmp(&other.weight).reverse(),
            other => other,
        })
    }
}

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

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

    fn set_visited(self, v: u64) -> Solution<'a> {
        Solution { visited: v, ..self }
    }

    fn incr_visited(self) -> Solution<'a> {
        self.set_visited(self.visited + 1)
    }
}
```

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() || current.cost >= current.inst.b { return current }

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

        if w <= m {
            let x = include();
            if x.cost < x.inst.b {
                let y = exclude();
                max(x, y).set_visited(x.visited + y.visited)
```

```
            } else { x }
        }
        else { exclude() }
    }

    let empty = Solution { weight: 0, cost: 0, visited: 0, cfg: Default::default(), inst: self };
    go(&self.items, empty, 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 = *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 >= current.inst.b || current.cost + prices[i] <= best.cos
            return current
        }

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

        if w <= m {
            let x = include();
            if x.cost < x.inst.b {
                let y = exclude(x);
                Solution { visited: x.visited + y.visited, ..max(x, y) }
            } else { x }
        }
        else { exclude(best) }
    }

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

Dynamické programování

Kromě dvou zkoumaných algoritmů jsem implementoval ještě třetí, který je ovšem založen na dynamickém programování. Jeho časová složitost je $\Theta(nM)$, kde M je kapacita batohu. Vzhledem k parametrům vstupních dat v tomto úkolu zvládá i ZR40 vstupy extrémně rychle (všech 500 instancí pod 300 ms), s velkými hodnotami M by si ovšem neporadil. V tuto chvíli nelze použít, protože počítá jen (konstruktivní) cenu, nikoliv celé řešení v podobě objektu struktury `Solution` jako je tomu u ostatních dvou.

```
fn dynamic_programming(&self) -> u32 {
    let (m, b, items) = (self.m, self.b, &self.items);
    let mut next = Vec::with_capacity(m as usize + 1);
    next.resize(m as usize + 1, 0);
    let mut last = Vec::new();

    for i in 1..=items.len() {
        let (weight, cost) = items[i - 1];
        last.clone_from(&next);

        for cap in 0..=m as usize {
            next[cap] = 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] + cost)
                };
        }
    }

    *next.last().unwrap() //>= b
}
```

Závěr

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

Appendix

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

```
use std::{io::stdin, str::FromStr, cmp, cmp::max};
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.

```
<<boilerplate>>

fn parse_line<T>(mut stream: T) -> Result<Option<Instance>> where T: std::io::BufRead {
    let mut input = String::new();
    match 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  b = 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, b, items}))
}
```

Výběr algoritmu je řízen argumentem předaným na příkazové řádce. Příslušnou funkci vrátíme jako hodnotu tohoto bloku:

```
let args: Vec<String> = std::env::args().collect();
if args.len() == 2 {
    let ok = |x: fn(&Instance) -> Solution| Ok(x);
    match &args[1][..] {
        "bf"    => ok(Instance::brute_force),
        "bb"    => ok(Instance::branch_and_bound),
        // "dp"    => ok(Instance::dynamic_programming),
        invalid => Err(anyhow!("\"{}\" is not a known algorithm", invalid)),
    }
} else {
    println!(
        "Usage: {} <algorithm>, where <algorithm> is one of bf, bb, dp",
        args[0]
    );
    Err(anyhow!("Expected 1 argument, got {}", args.len() - 1))
}
```

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

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

impl Boilerplate for std::str::SplitWhitespace<'_> {
    fn parse_next<T: FromStr>(&mut self) -> Result<T>
```

```
      where <T as FromStr>::Err: std::error::Error + Send + Sync + 'static {
        let str = self.next().ok_or(anyhow!("unexpected end of input"))?;
        str.parse::<T>()
           .with_context(|| format!("cannot parse {}", str))
    }
}
```

Automatické testy

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

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

    impl Arbitrary for Instance {
        fn arbitrary(g: &mut Gen) -> Instance {
            Instance {
                id:    i32::arbitrary(g),
                m:     u32::arbitrary(g),
                b:     u32::arbitrary(g),
                items: Vec::arbitrary(g)
                           .into_iter()
                           .take(10)
                           .map(|(w, c): (u32, u32)| (w.min(10_000), c.min(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.b    .shrink().map(|b    | Instance {b,     ..(&data).clone()}))
                .chain(self.items.shrink().map(|items| Instance {items, ..data}))
                .collect();
            Box::new(chain.into_iter())
        }
    }

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

            println!("{} >= {}", c, b);
            // assert!(c >= b);

            let (weight, 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();

        println!("{} <= {}", weight, *m);
        assert!(weight <= *m);

        println!("{} == {}", cost, *c);
        assert_eq!(cost, *c);

        println!("{} == {}", weight, *w);
        assert_eq!(weight, *w);
    }
}

#[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, b: 0, items: vec![(1, 1), (1, 2), (0, 1)] };
    assert_eq!(i.branch_and_bound().cost, i.brute_force().cost)
}

#[test]
fn small_bb_is_correct() {
    let a = Instance {
        id: -10,
        m: 165,
        b: 384,
        items: vec![ (86,  744)
                   , (214, 1373)
                   , (236, 1571)
                   , (239, 2388)
                   ],
    };
    a.branch_and_bound().assert_valid(&a);
}

#[test]
fn bb_is_correct() -> Result<()> {
    use std::fs::File;
    use std::io::BufReader;
    let inst = parse_line(
        BufReader::new(File::open("ds/NR15_inst.dat")?)
    )?.unwrap();
    println!("testing {:?}", inst);
    inst.branch_and_bound().assert_valid(&inst);
    Ok(())
}
```

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