NI-KOP – úkol 2

Ondřej Kvapil

Kombinatorická optimalizace: problém batohu

Zadání

- Na sadách instancí (NK, ZKC, ZKW) experimentálně vyhodnoťte závislost výpočetního času a u všech heuristických algoritmů také relativní chyby (průměrné i maximální) na velikosti instance následujících algoritmů pro
  - konstruktivní verzi problému batohu:
  - Metoda větví a hranic.
  - Metoda dynamického programování (dekompozice podle kapacity nebo podle cen),
  - Jednoduchá greedy heuristika
  - Modifikace této heuristiky (redux), která uvažuje také řešení se sólo nejdražší věcí
  - FPTAS algoritmem, tj. s použitím modifikovaného dynamického programování s dekompozicí podle ceny (při použití dekompozice podle kapacity není algoritmus FPTAS)
- Experiment má odpovědět na tyto otázky:
  - Odpovídají obě závislosti (kvality a času) předpokladům?
  - Je některá heuristická metoda systematicky lepší (tzv. dominance) v některém kritériu?
  - Jak se liší obtížnost jednotlivých sad z hlediska jednotlivých metod?
  - Jaká je závislost maximální chyby (ε) a času FPTAS algoritmu na zvolené přesnosti? Odpovídá předpokladům?

Pokyny

Algoritmy naprogramujte, využijte části programů z minulé úlohy.

Metodu větví a hranic použijte tak, aby omezujícím faktorem byla hodnota optimalizačního kritéria. Tj. použijte ořezávání shora (překročení kapacity batohu) i zdola (stávající řešení nemůže být lepší než nejlepší dosud nalezené).

Pozor! Pokud implementujete FPTAS pomocí zanedbávání bitů, musíte pro daný počet zanedbaných bitů vypočítat max. chybu (ε). V experimentálních výsledcích by počet zanedbaných bitů neměl figurovat, neb neříká nic konkrétního o přesnosti. Pozor, tato max. chyba je jiná pro každou instanci, nezávisí pouze na velikosti instance, ale také na max. ceně.

Pozor! V některých instancích se objevují věci, které svojí hmotností překračují kapacitu batohu. Samozřejmě se jedná o platné instance. Nicméně tyto věci komplikují přepočet cen u FPTAS. Zvažte sami, jak se s tím vypořádat. Řešení je snadné.

Pozn.: u této úlohy je opravdu lepší měřit skutečný CPU čas, namísto počtu konfigurací, jak tomu bylo u předchozího úkolu. Srovnávají se zde principiálně velice odlišné algoritmy, najít jiný relevantní způsob měření složitosti by bylo obtížné (ne-li nemožné).

Zpráva bude obsahovat

- ☐ Popis implementovaných metod.
- ☐ Srovnání výpočetních časů metody větví a hranic, dynamického programování a heuristiky cena/váha (stačí jedna). Grafy vítány.
  - Tj. závislosti výpočetních časů na velikosti instance
- ☐ Porovnání relativních chyb (průměrných a maximálních) obou heuristik.
  - Tj. závislosti rel. chyby na velikosti instance
- ☐ U FPTAS algoritmu pozorujte (naměřte, zdokumentujte) závislost chyby a výpočetního času algoritmu na zvolené přesnosti zobrazení (pro několik různých přesností), srovnání maximální naměřené chyby s teoreticky předpokládanou.
  - Tj. zvolte několik požadovaných přesností (ε), v závislosti na ε měřte čas běhu a reálnou (maximální, případně i průměrnou) chybu algoritmu
- ☐ Zhodnocení naměřených výsledků.

Bonusový bod

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

- Srovnání různých dekompozic v dynamickém programování (podle váhy, podle kapacity)
- detailní experimentální analýza FPTAS algoritmu,
- atd.

Řešení

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

Instrukce k sestavení programu

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

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

Benchmarking

Pro provedení měření výkonu programu jsem využil nástroje Hyperfine.

```
uname -a
./cpufetch --logo-short --color ibm
mkdir -p docs/measurements/
cd solver
hyperfine --export-json ../docs/bench.json \
          --parameter-list n 4,10,15,20,22,25 \
          --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	**102**.63 ms	**2**.14 ms	**99**.8 ms	**102**.17 ms	**110**.33 ms
bf	N	10	**106**.18 ms	**1**.74 ms	**104**.21 ms	**105**.7 ms	**111**.13 ms
bf	N	15	**225**.31 ms	**3**.74 ms	**221**.43 ms	**223**.11 ms	**232**.81 ms
bf	N	20	**4451**.14 ms	**120**.46 ms	**4379**.95 ms	**4397**.11 ms	**4630**.41 ms
bf	N	22	**15006**.52 ms	**571**.57 ms	**14712**.19 ms	**14725**.03 ms	**15863**.81 ms
bf	N	25	**134776**.9 ms	**15140**.56 ms	**124465**.59 ms	**129038**.38 ms	**156565**.27 ms
bf	Z	4	**104**.38 ms	**2**.44 ms	**101**.54 ms	**104**.03 ms	**113**.98 ms
bf	Z	10	**98**.5 ms	**3**.68 ms	**95**.36 ms	**97**.71 ms	**109**.88 ms
bf	Z	15	**263**.6 ms	**4**.86 ms	**260**.04 ms	**261**.78 ms	**274**.85 ms
bf	Z	20	**5865**.8 ms	**8**.06 ms	**5857**.27 ms	**5865**.32 ms	**5875**.3 ms
bf	Z	22	**29251**.71 ms	**10133**.34 ms	**22793**.9 ms	**24927**.96 ms	**44357**.03 ms
bf	Z	25	**211507**.97 ms	**8518**.15 ms	**204125**.43 ms	**211501**.11 ms	**218904**.21 ms
bb	N	4	**91**.18 ms	**2**.93 ms	**88**.46 ms	**90**.38 ms	**103**.99 ms
bb	N	10	**94**.48 ms	**4**.97 ms	**89**.8 ms	**92**.04 ms	**104**.87 ms
bb	N	15	**104**.54 ms	**1**.16 ms	**102**.54 ms	**104**.47 ms	**107**.06 ms
bb	N	20	**388**.98 ms	**7**.61 ms	**384**.22 ms	**387**.09 ms	**405**.91 ms
bb	N	22	**1264**.78 ms	**16**.52 ms	**1253**.28 ms	**1258**.36 ms	**1289**.13 ms
bb	N	25	**7562**.55 ms	**57**.7 ms	**7523**.88 ms	**7539**.71 ms	**7646**.91 ms
bb	Z	4	**91**.1 ms	**1**.31 ms	**89**.41 ms	**90**.85 ms	**95**.6 ms
bb	Z	10	**94**.17 ms	**1**.45 ms	**91**.91 ms	**93**.88 ms	**97**.63 ms
bb	Z	15	**154**.63 ms	**3**.13 ms	**151**.52 ms	**153**.1 ms	**162**.26 ms
bb	Z	20	**1595**.75 ms	**0**.7 ms	**1594**.97 ms	**1595**.8 ms	**1596**.44 ms
bb	Z	22	**5341**.89 ms	**112**.87 ms	**5279**.97 ms	**5288**.26 ms	**5511**.08 ms
bb	Z	25	**37373**.11 ms	**761**.48 ms	**36775**.09 ms	**37158**.28 ms	**38400**.79 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.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")
```

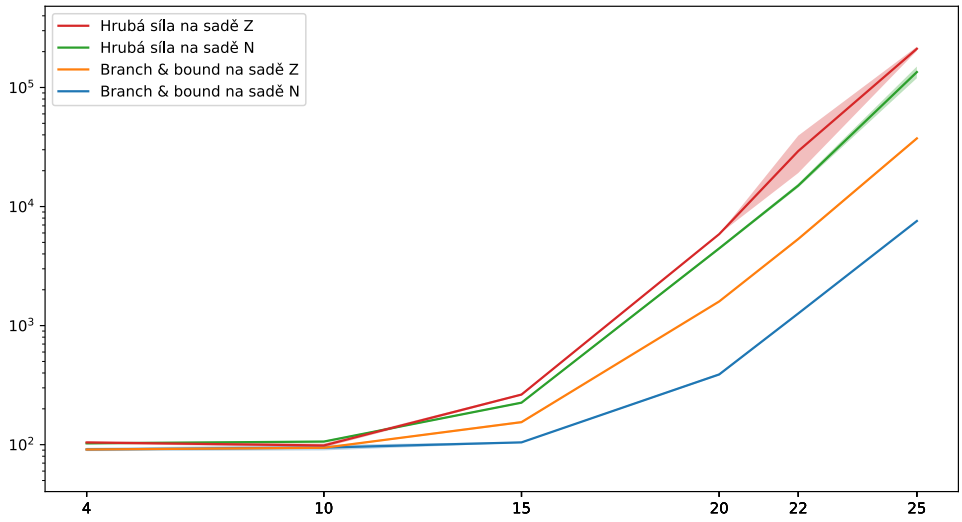

Figure 1: Závislost doby běhu na počtu předmětů. Částečně průhledná oblast značí směrodatnou odchylku (σ).

```
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')
```

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

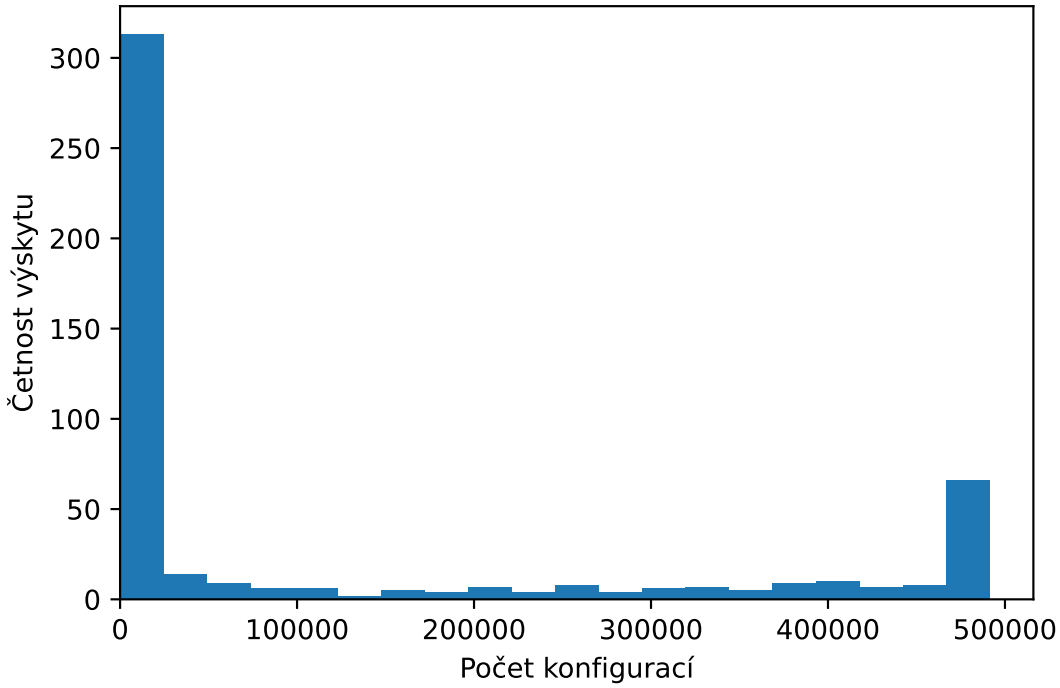

Figure 2: Sada NR: Hrubá síla pro $n = 15$

Analýza

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

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

Implementace

Program začíná definicí datové struktury reprezentující instanci problému batohu.

```
#[derive(Debug, PartialEq, Eq, Clone)]
struct Instance {
```

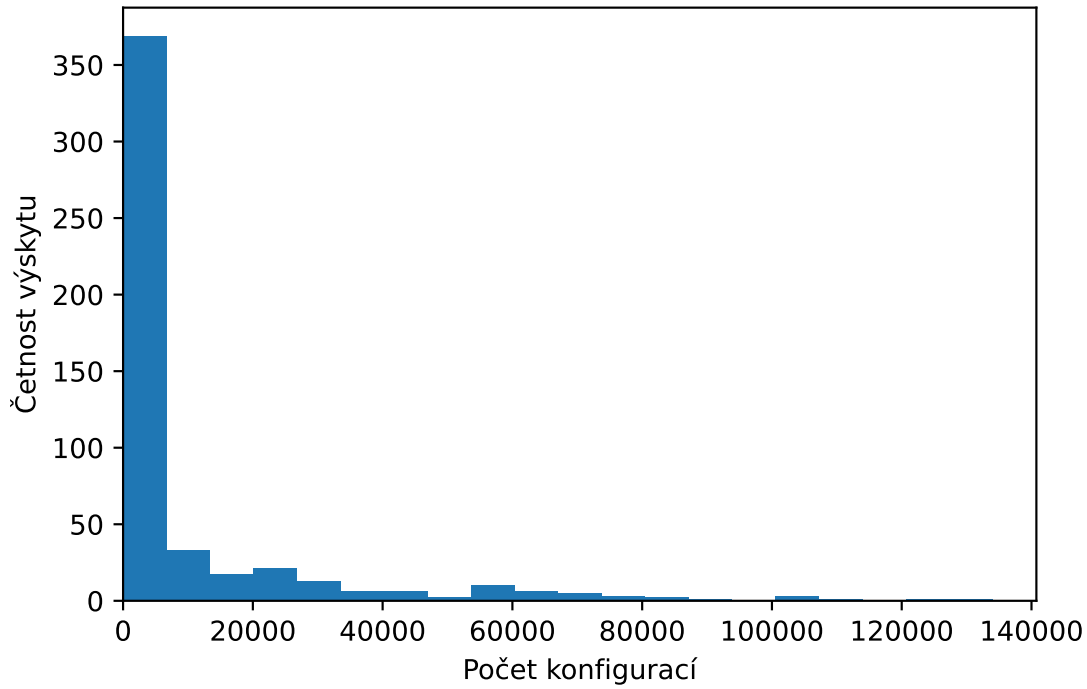

Figure 3: Sada NR: Metoda větví a hranic pro $n = 15$

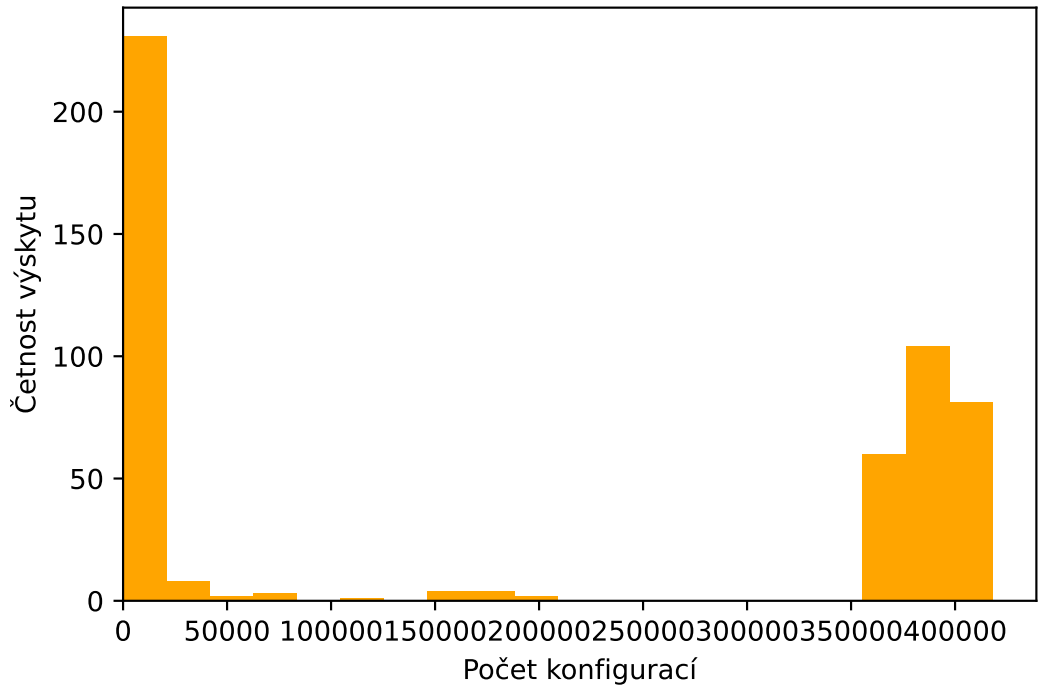

Figure 4: Sada ZR: Hrubá síla pro $n = 15$

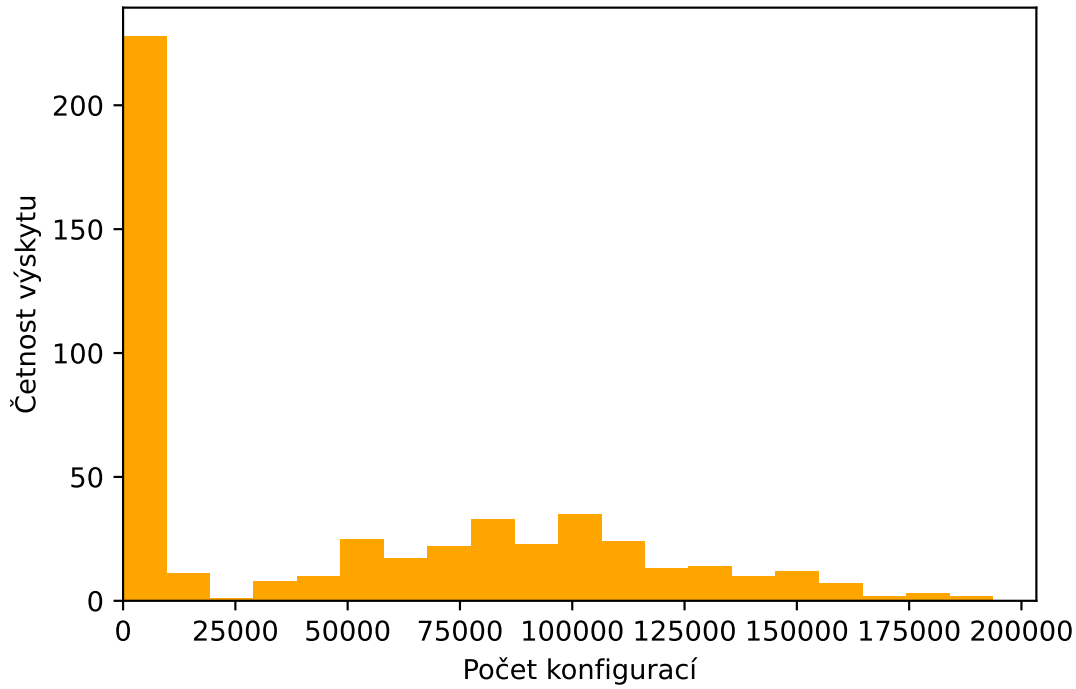

Figure 5: Sada ZR: Metoda větví a hranic pro $n = 15$

```
    id: i32, m: u32, b: u32, items: Vec<(u32, u32)>
}
```

Následující úryvek poskytuje ptačí pohled na strukturu souboru. Použité knihovny jsou importovány na začátku, následuje již zmíněná definice instance problému, dále funkce `main()`, parser, definice struktury řešení a její podpůrné funkce, samotné algoritmy řešiče a v neposlední řadě sada automatických testů.

```
<<imports>>

<<problem-instance-definition>>

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

    loop {
        match parse_line(stdin().lock())?.as_ref().map(alg) {
            Some(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.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 += 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.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![0; m as usize + 1];
    let mut last = vec![];

    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>>
```

```rust
fn parse_line<T>(mut stream: T) -> Result<Option<Instance>> where T: std::io::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  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:

```rust
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.

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

impl Boilerplate for std::str::SplitWhitespace<'_> {
    fn parse_next<T: FromStr>(&mut self) -> Result<T>
      where <T as FromStr>::Err: std::error::Error + Send + Sync + 'static {
        let str = self.next().ok_or_else(|| anyhow!("unexpected end of input"))?;
        str.parse::<T>()
```

```
            .with_context(|| format!("cannot parse {}", str))
    }
}
```

Automatické testy

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

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

    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);
    }
}
```