

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 soubor obsahuje veškerý zdrojový kód nutný k reprodukci méj práce.

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 \
  --parameter-list alg bf,bb \
  --runs 2 \
  --style color \
  'cargo run --release -- {alg} \
    < ds/NR{n}_inst.dat \
    > ../docs/measurements/{alg}-{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 "algoritmus, \$n, \$průměr, \$\pm \sigma, \$minimum, medián, maximum" > docs/bench.csv
jq -r \
  '.[] | .[] | [.parameters.alg, .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.

algoritmus	n	průměr	$\pm\sigma$	minimum	medián	maximum
bf	4	106.56 ms	6.39 ms	102.04 ms	106.56 ms	111.08 ms
bf	10	112.95 ms	4.51 ms	109.76 ms	112.95 ms	116.14 ms
bf	15	323.27 ms	0.22 ms	323.12 ms	323.27 ms	323.43 ms
bb	4	104.23 ms	3 ms	102.11 ms	104.23 ms	106.35 ms
bb	10	104.45 ms	0.68 ms	103.97 ms	104.45 ms	104.93 ms
bb	15	139.04 ms	3.67 ms	136.44 ms	139.04 ms	141.64 ms

Srovnání algoritmů

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

df = pd.read_csv("docs/bench.csv", dtype = "string")
df.rename({
    "algoritmus": "alg",
    "$n$": "n",
    "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"]
df[numeric_columns] = df[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 df.groupby("alg"):
    (x, y, sigma) = (group["n"], group["avg"], group["sigma"])
    ax.plot(x, y, label = labels[name])
    ax.fill_between(x, y + sigma, y - sigma, alpha = 0.3)

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

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

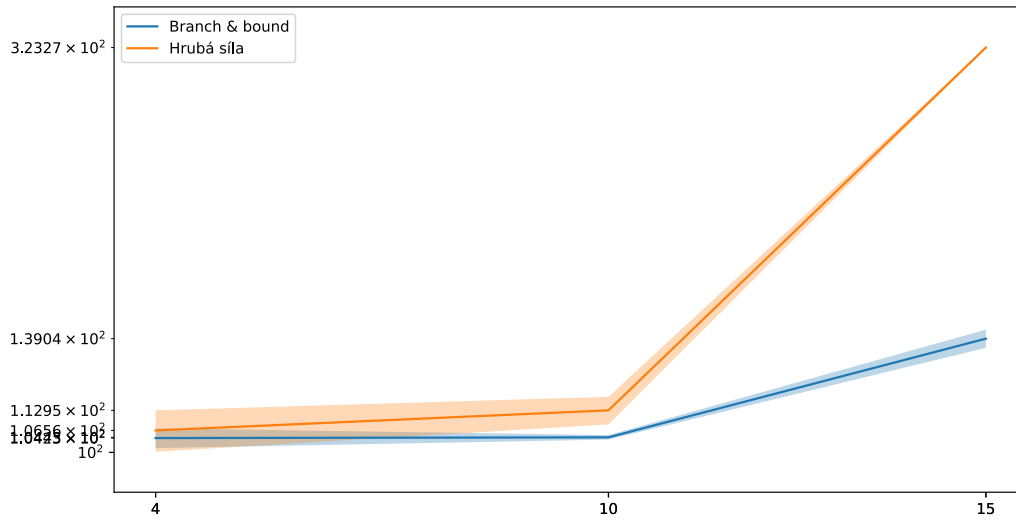


Figure 1: Závislost doby běhu na počtu předmětů. Částečně průhledná oblast značí směrodatnou odchylku (σ).

Implementace

Program začíná definicí datové struktury reprezentující instanci problému batohu.

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

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;

<<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>>

#[inline(always)]
fn smart_max<A, F, G>(f: F, g: G) -> A
where F: Fn() -> A
    , G: Fn(A) -> A
    , A: cmp::Ord + Copy {
    let x = f();
    max(x, g(x))
}

#[inline]
fn smart_max<'a, F, G>(f: F, g: G) -> Solution<'a>
where F: Fn() -> Solution<'a>
    , G: Fn(Solution) -> Solution {
    let x = f();
    let y = g(x);
    Solution { visited: x.visited + y.visited, ..max(x, y) }
}

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 }

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)
}
}

impl Instance {
    <<solver-dp>>

    <<solver-bb>>

    <<solver-bf>>
}

<<tests>>

```

Algorithmy

Hrubá síla

```

fn brute_force(&self) -> u32 {
    self.brute_force2().cost
}

fn brute_force_old(&self) -> u32 {
    let (m, b, items) = (self.m, self.b, &self.items);
    fn go(items: &Vec<(u32, u32)>, cap: u32, i: usize) -> u32 {
        if i >= items.len() { return 0; }

        let (w, c) = items[i];
        let next = |cap| go(items, cap, i + 1);
        let include = || next(cap - w);
        let exclude = || next(cap);
        if w <= cap {
            max(c + include(), exclude())
        } else {
            exclude()
        }
    }

    go(items, m, 0)
}

```

```

fn brute_force2(&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.clone().with(i).incr_visited();
            next(current, m - w)
        };
        let exclude = || next(current.incr_visited(), m);

        if w <= m {
            let x = include();
            let y = exclude();
            max(x, y).set_visited(x.visited + y.visited)
        }
        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) -> u32 {
    self.branch_and_bound2().cost
}

fn branch_and_bound2(&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) -> Solution<'a> {
        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.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();
        let y = exclude(x);
        Solution { visited: x.visited + y.visited, ..max(x, y) }
    }
    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

```

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
    }
}

```

Appendix

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_force2),
        "bb"    => ok(Instance::branch_and_bound2),
        // "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.

```
#[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(), i.brute_force())
}

#[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_bound2().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_bound2().assert_valid(&inst);
    Ok(())
}

#[quickcheck]
fn qc_bb_is_really_correct(inst: Instance) {
    assert_eq!(inst.branch_and_bound2().cost, inst.brute_force());
}
}

```