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 mojí 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
cd solver
hyperfine --export-json ../docs/bench.json \
          --parameter-list n 4,10,15,20,22,25,27,30,32 \
          --parameter-list alg bf,bb,dp \
          --min-runs 4 \
          --style color \
          'cargo run --release -- {alg} < ds/NR{n}_inst.dat' 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 ""
```

algoritmus	n	průměr	$\pm\sigma$	minimum	medián	maximum

+==============+=======+==========+================+==========+======
+————————+———-+————-+————————-+————————+————-+————————+

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")
```

Implementace

Program začíná definicí datové struktury reprezentující instanci problému batohu.

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

use std::{io::stdin, str::FromStr};
use anyhow::{Context, Result, anyhow};

<<problem-instance-definition>>
```

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

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

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

<<parser>>

impl Instance {
    <<solver-dp>>

    <<solver-bb>>

    <<solver-bf>>
}
```

Algoritmy

Hrubá síla

```rust
fn brute_force(&self) -> u32 {
    let (m, b, items) = (self.m, self.b, &self.items);
    fn go(items: &Vec<(u32, u32)>, cap: u32, i: usize) -> u32 {
        use std::cmp::max;
        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);
        let current = if w <= cap {
            max(c + include(), exclude())
        } else {
            exclude()
        };
        current
    }

    go(items, m, 0)
}
```

Branch & bound

```rust
fn branch_and_bound(&self) -> u32 {
    let Instance { m, b, items, .. } = self;
    let prices: Vec<u32> = items.iter().rev()
        .scan(0, |sum, (_w, c)| {
            *sum = *sum + c;
            Some(*sum)
        })
        .collect::<Vec<_>>().into_iter().rev().collect();

    struct State<'a>(&'a Vec<(u32, u32)>, Vec<u32>);
    fn go(state: &State, best: u32, cap: u32, i: usize) -> u32 {
        let State(items, prices) = state;
        if i >= items.len() || best > prices[i] { return 0; }

        let (w, c) = items[i];
        let next = |best, cap| go(state, best, cap, i + 1);
        let include = || next(best, cap - w);
        let exclude = |best| next(best, cap);
        if w <= cap {
            use std::cmp::max;
            let new_best = max(c + include(), best);
            max(new_best, exclude(new_best))
        } else {
            exclude(best)
        }
    }
```

```
    go(&State(items, prices), 0, *m, 0)
}
```

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 {
                use std::cmp::max;
                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() -> Result<Option<Instance>> {
    let mut input = String::new();
    match stdin().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) -> u32| 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))
    }
}
```