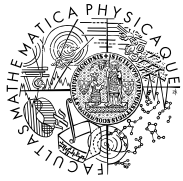


MATEMATICKO-FYZIKÁLNÍ
FAKULTA
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Jiří Kalvoda

Binární paint shop problém

Informatický ústav Univerzity Karlovy

Vedoucí bakalářské práce: doc. Mgr. Robert Šámal, Ph.D.
Studijní program: Informatika

Praha 2024

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů. Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ....

Podpis autora

Název práce: Binární paint shop problém

Autor: Jiří Kalvoda

Ústav: Informatický ústav Univerzity Karlovy

Vedoucí bakalářské práce: doc. Mgr. Robert Šámal, Ph.D., Informatický ústav Univerzity Karlovy

Abstrakt: Binární paint shop je následující optimalizační úloha: Na barvicí linku vjíždí řada aut. Od každého typu auta jsou někde v řadě právě dvě auta a jedno z nich bychom rádi nabarvili červeně a druhé modře. Měnit barvu, kterou aktuálně barvíme, je drahá operace, proto bychom rádi pro danou posloupnost aut provedli co nejméně změn. Vstupem úlohy jsou typy aut v posloupnosti a výstupem je jejich obarvení. Je známé, že úloha je NP-těžká a za určitých předpokladů dokonce neaproximovatelná, proto je na místě zkoumat řešení, co se chovají dobře na náhodném vstupu. V této práci je představen algoritmus založený na semidefinitním programování, který dle provedených měření pro náhodné vstupy dosahuje výsledků okolo 0.34-násobku počtu typů aut. O algoritmu jsme dokázali, že pro každý vstup vrátí ve střední hodnotě řešení nejhůře o 0.212-násobek počtu typů aut horší než optimum.

Klíčová slova: binární paint shop problém, aproximační algoritmus

Title: Binary paint shop problem

Author: Jiří Kalvoda

Institute: Computer Science Institute of Charles University

Supervisor: doc. Mgr. Robert Šámal, Ph.D., Computer Science Institute of Charles University

Abstract: The Binary Paint Shop represents an optimization problem: A line of cars enters a paint shop. There are exactly two cars of each type somewhere in the line. The aim is to color one of these two cars red and the other blue. It is expensive to switch the current color in the paint shop so for a given sequence of cars we would like to minimize the number of color changes. Input of the task are types of cars in the line and output is the coloring of theses cars. This task is known to be NP-hard, and under specific conditions, it defies polynomial time approximations. Therefore, it is a good idea to find some algorithms which behave well on a randomly generated input. This thesis introduces an algorithm based on semidefinite programming. Experiments on random inputs show it reaches solutions near to 0.34 times the number of car types. We proved that for each input this algorithm returns a solution with expected deviation from optimum of at most 0.212 times the number of car types.

Keywords: binary paint shop problem, approximation algorithm

Obsah

1 Úvod

V této práci představíme binární paint shop problem. Jedná se o úlohu, kterou neumíme efektivně řešit (protože je NP úplná) a ani aproximovat (za předpokladu Unique games conjecture je konstantní aproximace také NP těžká), takže mimo jiné probíhá aktivní výzkum snažící se najít algoritmus, který je dobrý v průměrném případě (pro náhodný vstup).

1.1 Zadání

Zadání binárního paint shop problému (dále též BPS) je následující:

> **Úloha (Binární paint shop):** V řadě je $2n$ aut n různých typů – od každého typu dvě. Chtěli bychom od každého typu nabarvit jedno auto červeně a druhé modře. Auta však na barvicí linku vjíždí v pořadí, v jakém jsou v řadě. Barvicí linka je optimalizovaná na barvení velkého počtu aut jednou barvou. Tedy měnit barvu, kterou se barví, je složitá a drahá záležitost. Chceme tedy najít obarvení aut tak, aby od každého typu bylo jedno červené a jedno modré, přitom počet změn barev v řadě byl co nejmenší.

Počet změn barev řešení považujeme za *skóre* algoritmu. *Relativní skóre* pak je poměr skóre a počtu typů aut.

Problém můžeme chápat buď jako optimalizační problém, kde účelová funkce je počet změn v řešení a snažíme se ji minimalizovat, nebo jako rozhodovací problém, kde se ptáme, jestli existuje řešení s nejvýše nějakým zadaným počtem změn.

1.2 Cíle práce

Naším cílem bude najít co nejlepší algoritmus řešící BPS a odhadnout u něj střední hodnotu skóre pro náhodný vstup. Dále se pokusíme porovnat již známé algoritmy.

1.3 Struktura práce

Nejprve zavedeme notaci potřebnou pro pohodlnou práci s BPS a definujeme aproximační algoritmy. Kapitola 2 obsahuje shrnutí doposud známých algoritmů a jiných výsledků ohledně BPS. V kapitole 3 je představen princip semidefinitního programování, které má uplatnění v algoritmu představeném o kapitole 4. Kapitola 5 se věnuje praktické implementacím algoritmů a naměřeným datům o nich.

1.4 Notace

Auta budeme indexovat od 0 do $2n - 1$ a typy od 0 do $n - 1$.

Pomocí $a_{i,0}$ budeme značit pozici prvního auta typu i a pomocí $a_{i,1}$ pozici druhého.

Opačně t_i bude značit typ auta na pozici i a p_i bude značit, jestli se jedná o první nebo druhé auto daného typu (tedy bude nabývat hodnoty 0 nebo 1).

Protože někdy bude výhodnější zacházet se znaménky, zavedeme $P_i = -1 + 2p_i$, které bude nabývat hodnot -1 a 1. Také rozšíříme notaci o $a_{i,-1} = a_{i,0}$.

Nakonec ještě zavedeme zkratku indexu druhého auta stejného typu jako je auto na pozici i: $o_i = a_{t_i,-P_i}$.

Barvy pro nás budou 0 a 1. Pokud budeme uvažovat nějaké konkrétní obarvení c, tak $c(i)$ bude značit barvu auta na pozici i.

Pro algoritmus **alg** označíme skóre na vstupu α pomocí $\gamma_{\mathbf{alg}}(\alpha)$. Dále označme všechny vstupy délky n jako W_n. Průměrné skóre algoritmu na všech vstupech délky n tedy bude

$$\gamma_{\mathbf{alg}}(n) = \frac{\sum_{\alpha \in W_n} \gamma_{\mathbf{alg}}(\alpha)}{|W_n|}.$$

Tuto hodnotu mimo jiné můžeme chápat jako střední hodnotu skóre pro uniformně náhodný vstup délky n. K tomu ještě zavedeme notaci pro relativní skóre $\delta_{\mathbf{alg}}(\alpha) = \gamma_{\mathbf{alg}}(\alpha)/n_\alpha$ a analogicky průměrné relativní skóre $\delta_{\mathbf{alg}}(n) = \gamma_{\mathbf{alg}}(n)/n$. Nakonec označme $\delta_{\mathbf{alg}} = \lim_{n\to\infty} \delta_{\mathbf{alg}}(n) = \lim_{n\to\infty} \gamma_{\mathbf{alg}}(n)/n$. Protože limita nemusí vždy existovat, zavedeme ještě limes superior a limes inferior: $\delta^+_{\mathbf{alg}} = \limsup_{n\to\infty} \delta_{\mathbf{alg}}(n)$ a $\delta^-_{\mathbf{alg}} = \liminf_{n\to\infty} \delta_{\mathbf{alg}}(n)$. Význam těchto definic bude vysvětlen v následující kapitole.

Dále označme $\gamma(\alpha)$ optimální skóre na vstupu α. A následně analogicky definujeme $\gamma(n), \delta(\alpha), \delta(n)$, δ, δ^+ a δ^- stejně jako u variant s algoritmem, jen skóre daného algoritmu nahradíme za optimální skóre.

1.5 Definice aproximačních algoritmů

Protože ne pro všechny problémy známe polynomiální algoritmus, který je schopný je vyřešit, zajímavý výsledek může být, i když se k řešení zvládneme jen v nějakém smyslu alespoň přiblížit. Na to nejprve musíme říct, co pro nás znamená, že nějaké řešení je několikrát horší než jiné. To nám poskytne obecná definice optimalizačního problému.

Definice (Optimalizační problém): Problém je *optimalizační*, pokud pro každý vstup I, existuje množina přípustných řešení $F(I)$. Dále existuje účelová funkce f, která pro každý vstup a jeho přípustné řešení určuje reálné nezáporné číslo – jeho hodnotu.

Pokud se jedná o minimalizační problém, tak pod pojmem *optimum* daného vstupu (značíme $\mathrm{opt}(I)$) myslíme infimum hodnot účelové funkce přes všechna přípustná řešení, tedy $\inf f[F(i)]$. Pro maximalizační problém analogicky použijeme supremum.

A nyní již přejdeme k samotným definicím algoritmů blížících se optimu.

Definice (Aproximační algoritmus): Algoritmus **alg** je $g(n)$-aproximační na minimalizačním problému, pokud pro každý vstup I algoritmus vrátí přípustné řešení $\mathbf{alg}(I)$, pro které platí, že $f(\mathbf{alg}(I)) \leq g(|I|) \cdot \mathrm{opt}(I)$.

Předešlá definice záleží na tom, co považujeme za velikost vstupu, což se pro různé problémy a jejich interpretace liší. Naštěstí nás většinou budou zajímat c-aproximace, tedy aproximace, kde $g(n)$ je konstantní funkce rovna c.

U maximalizačních problémů je drobný problém v terminologii, protože není shoda na tom, jestli má platit $f(\mathbf{alg}(I)) \geq g(|I|) \cdot \mathrm{opt}(I)$ nebo $f(\mathbf{alg}(I)) \geq \frac{1}{g(|I|)} \cdot \mathrm{opt}(I)$. Naštěstí podle kontextu jde snadno rozhodnout, která definice se

používá, protože je potřeba násobit optimum číslem menším rovno jedné (jinak by pro kladné hodnoty účelové funkce nemohl existovat žádný vyhovující algoritmus).

Bohužel někdy asi ani s aproximačními algoritmy nevystačíme a proto zavedeme pravděpodobnostní relaxaci.

Definice (Pravděpodobnostní aproximační algoritmus): Náhodný algoritmus **alg** je pravděpodobnostně $g(n)$-aproximační na minimalizačním problému, pokud pro každý vstup I algoritmus vždy vrátí přípustné řešení $\mathbf{alg}(I)$, pro které platí, že $\mathbb{E}[f(\mathbf{alg}(I))] \leq g(|I|) \cdot \mathrm{opt}(I)$.

Snadným důsledkem definice (a Markovovy nerovnosti) je, že pro každé $\lambda > 1$ pravděpodobnostně $g(n)$-aproximační algoritmus vrátí s pravděpodobností zdola omezenou konstantou $1 - \frac{1}{\lambda} \in (0, 1)$ řešení s hodnotou nejvýše $\lambda \cdot g(n) \cdot \mathrm{opt}(I)$.

Všimněme si, že na hodnotě $1 - \frac{1}{\lambda}$ příliš nezáleží. Opakovaným spouštěním algoritmu a pak vybráním nejlepšího z dodaných řešení zvládneme pravděpodobnost libovolně těsně přiblížit k jedné.

Pokud řešíme složitost algoritmů a úloh, většinou vyžadujeme, aby účelová funkce i rozhodování přípustnosti řešení byly vyčíslitelné v polynomiálním čase. Navíc chceme, aby všechna přípustná řešení měla délku omezenou nějakým polynomem v délce vstupu. Snadno nahlédneme, že za takovýchto podmínek je rozhodovací verze, jestli je optimum alespoň zadané číslo, v NP. Pro takovou úlohu většinou hledáme polynomiální aproximační algoritmus. Binární paint shop vyhovuje všem těmto podmínkám.

Pokud existuje $(1+\varepsilon)$-aproximační algoritmus pro každé $\varepsilon > 0$, říkáme, že máme aproximační schéma.

2 Doposud známé výsledky

Bonsmaa, Epping a Hochstättler [1] dokázali, že optimalizační verze BPS je APX-těžký problém, což je silnější tvrzení, než že rozhodovací verze je NP-těžká. Za předpokladu P $\neq$ NP víme, že kromě polynomiálního algoritmu na rozhodovací verzi nemůže existovat ani polynomiální aproximační schéma na optimalizační verzi.

Snadno nahlédneme, že rozhodovací problém patří do NP. Když nám někdo dá přiřazení barev autům, v lineárním čase jsme schopni ověřit, že počet změn je dostatečně malý a od každého typu a barvy auta je právě jeden výskyt. Tedy rozhodovací problém je NP-úplný.

Dále pak Gupta a spol. [2] ukázali, že za předpokladu Unique games conjecture je NP-těžké i problém libovolně konstantně aproximovat. K tomu využili převod na problém minimálního ne-řezu. To je problém podobný maximálnímu řezu, který bude představen později. Ovšem místo maximalizace počtu hran v řezu minimalizujeme počet hran mimo něj (tedy v rámci partit). Tyto dva problémy mají stejné optimum, ovšem aproximovatelnost se na nich chová drasticky jinak. V momentě, kdy jsou skoro všechny hrany v řezu, tak přidání další hrany do řezu skoro nezmění počet hran v něm. Ovšem odebrání hrany z ne-řezu bude mít velký vliv (procentuálně) na počet hran v ne-řezu.

Vzhledem k předešlým výsledkům je na místě zkoumat různé heuristiky a obecně algoritmy, co jsou nás schopny k optimálnímu řešení alespoň částečně přiblížit.

Jedním z takových algoritmů je hladový algoritmus popsaný Andresem and Hochstättlerem [3]:

Algoritmus (Hladový algoritmus, g): Autům budeme přiřazovat barvy v pořadí, v jakém jsou na vstupu. První auto v řadě obarvíme řekněme červeně. Dále budeme vždy první auto daného typu barvit stejně jako předcházející auto a druhé auto daného typu obarvíme vždy zbývající barvou.

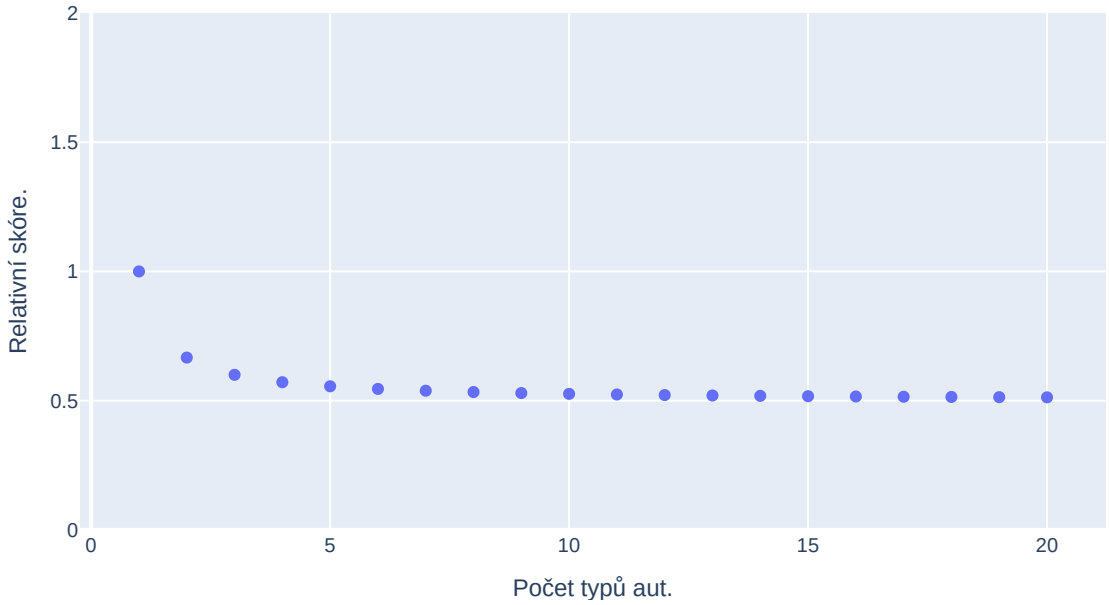
V daném článku autoři o tomto algoritmu dokázali, když vezmeme uniformně náhodný vstup délky n, tak střední hodnota počtu změn ve vygenerovaném řešení je $\sum_{0\leq k<n} \frac{2k^2-1}{4k^2-1}$.

Hladový algoritmus budeme značit $\mathbf{g}$, tedy předešlou hodnotu značíme $\gamma_{\mathbf{g}}(n)$. Připomeňme, že pomocí $\delta_{\mathbf{g}}(n)$ značíme tuto hodnotu vydělenou velikostí vstupu, tedy $\gamma_{\mathbf{g}}(n)/n$.

Pro dostatečně velká n se $\gamma_{\mathbf{g}}(n)$ pohybuje zhruba okolo $(1/2)n$ (formálně řečeno: platí, že $\delta_{\mathbf{g}} = \lim_{n\to\infty} \gamma_{\mathbf{g}}(n)/n = 1/2$) (viz obrázek 1).

U hladového řešení je tedy střední hodnota skóre přes uniformně náhodný vstup přesně vyčíslená. Pro nás je zejména důležité, že jsme ji schopni shora odhadnout. Střední hodnotu totiž můžeme považovat za ukazatel kvality algoritmu (čím menší je, tím se jedná o lepší algoritmus) a tedy horní odhad nám dává záruku kvality algoritmu.

Zajímavé je pro nás zkoumat chování algoritmů na velkých vstupech, tedy dává smysl uvažovat limitu střední hodnoty do nekonečna. Ovšem s narůstajícím počtem aut narůstá i počet potřebných změn, takže samotná limita střední hodnoty moc nedává smysl, protože by byla nekonečná. Místo ní budeme uvažovat



Obrázek 1: Graf střední hodnoty rel. skóre hladového řešení $\delta_{\mathbf{g}}(n)$ v závislosti na n.

$\lim_{n\to\infty} \gamma_{\mathbf{alg}}(n)/n = \lim_{n\to\infty} \delta_{\mathbf{alg}}(n)$. O ní víme, že pro libovolný algoritmus (pokud existuje) bude v intervalu $[0, 2]$, protože maximální počet změn musí být mezi 0 a $2n - 1$.

I když nejsme schopní hledat optimum efektivně, pro každý vstup je určitě optimum dobře definovaná hodnota. Můžeme se tedy ptát na otázku, kolik vyjde limita, kde místo skóre algoritmu vložíme optimum pro daný vstup, tedy $\lim_{n\to\infty} \delta(n) = \lim_{n\to\infty} \gamma(n)/n$. Označme hodnotu této limity δ.

Bohužel není zřejmé, že limita skutečně existuje, proto místo limity budeme často uvažovat limes supervisor a inferior. Ty budeme značit přidáním $\pm$ jako horního indexu. Stejnou notaci budeme používat i u limit algoritmů.

Jelikož optimum musí být alespoň tak dobré jako výstup hladového řešení, dostáváme konstruktivní horní odhad $\delta^+ \leq 0.5$. Hledáním lepších algoritmů můžeme horní odhad zlepšovat.

Překvapivě však je, že Hančl a kol. [4] ukázali dolní odhad $\delta^- \geq 0.214$ pomocí počítání pravděpodobností pro náhodné obarvení. Samotný fakt, že $\delta^- > 0$ je již poměrně zajímavé tvrzení. To nám říká, že každý algoritmus na průměrném vstupu musí použít aspoň $\Omega(n)$ změn barev. Nemůže tedy například existovat algoritmus, kterému by stačilo jen $\mathcal{O}(n/\log n)$ změn. To také říká, že odhadovat algoritmy pomocí $\lim_{n\to\infty} \gamma_{\mathbf{alg}}/n$, je alespoň co se týče asymptotiky dostačující, protože vystihuje konstantu nejvýznačnějšího členu polynomiální aproximace.

Dále si představíme několik algoritmů, které se snaží konstruktivně zlepšit horní odhad na δ^+.

Algoritmus (Rekurzivní hladové řešení, rg): Z řady aut odstraníme první auto v řadě a druhé auto příslušného typu. Zbytek aut rekurzivně obarvíme a pak do řady přidáme odebranou dvojici tak, aby byl počet změn co možná nejmenší možný.

Autoři algoritmu, Andres and Hochstättler [3], o něm dokázali, že $\frac{2}{5}\,n - \frac{8}{15} \leq \gamma_{\mathbf{rg}}(n) \leq \frac{2}{5}\,n + \frac{7}{10}$, tedy $\delta_{\mathbf{rg}} = 2/5 = 0.4$.

Dále se o zlepšení tohoto algoritmu pokusili Hančl a kol. [4]. Ti si všimli, že ve výstupu rekurzivního řešení se občas stane, že přebarvením dvojic typů aut se zlepší skóre. Ukázali, že takových dvojic je vždy ve výstupu lineárně mnoho a z toho pak ukázali horní odhad $\delta^+ \leq 0.4 - \varepsilon < 0.4$ (pro ε zhruba $2 \cdot 10^{-6}$).

Hančl a kol. [4] přišli ještě s dalším způsobem, jak vylepšit rekurzivní hladové řešení. Při běhu hladového řešení se občas stane, že některá dvojice aut stejného typu si může prohodit barvy bez změny skóre řešení. Když přidáváme auto do okolí takovéto dvojice, v některých případech můžeme provést toto prohození a tím snížit počet nově vytvořených změn barev. Budeme si tedy udržovat některé z takovýchto dvojic a pokud budeme přidávat auto do okolí evidované dvojice tak, že prohození barev dané dvojice by pomohlo, prohodíme její barvy.

Budeme pracovat nad rozšířenou abecedou barev o znak „$*$“ reprezentující neurčenou barvu. Při rekurzi budeme udržovat invariant, že pro každý typ obě auta buď budou označena $*$ a nebo ani jedno z nich nebude označeno $*$ a pak nutně budou mít různé barvy. Navíc bude platit, že $*$ nikdy není na okrajích a nejsou dvě vedle sebe. Specificky tedy na $*$ budeme přebarvovat dvojici aut v momentě, kdy získá všechny sousedy.

Pro popis algoritmu zavedeme notaci $N(i)$, což bude reprezentovat multimnožinu barev aut sousedících s autem na pozici $0 < i < 2n - 1$, tedy $\{c(i-1), c(i+1)\}$.

Algoritmus (Hvězdičkové rekurzivní řešení, rsg): Budeme pracovat nad rozšířenou škálou barev $\{0, 1, *\}$. Ze vstupu odebereme auta typu t_0 a zarekurzíme se na zbytek. Tím získáme nějaké obarvení, které může použít na původní vstup s tím, že auta typu t_0 zatím nebudou obarvená, ty dobarvíme dle následujících pravidel:

A) Když $o_0 = 1$ a $n > 1$ nastavíme $c(1) = c(2)$.
B) Když $o_0 = 2n - 1$ nastavíme $c(0) = 1$.
C) Když $N(o_0) = \{t, t\}$ pro nějaké $t \in \{0, 1\}$, nastavíme i $c(o_0) = t$.
D) Když $N(o_0) = \{0, 1\}$, nastavíme $c(0) = c(1)$.
E) Když $N(o_0) = \{1 - c(1), *\}$, nastavíme $c(0) = c(1)$.
F) Když $N(o_0) = \{c(1), *\}$, nastavíme $c(0) = c(1)$ a přenastavíme $*$ v okolí o_0 na $1 - c(1)$ a druhé auto daného typu na opačnou barvu.

A druhé auto typu t_0 obarvíme zbývající barvou.

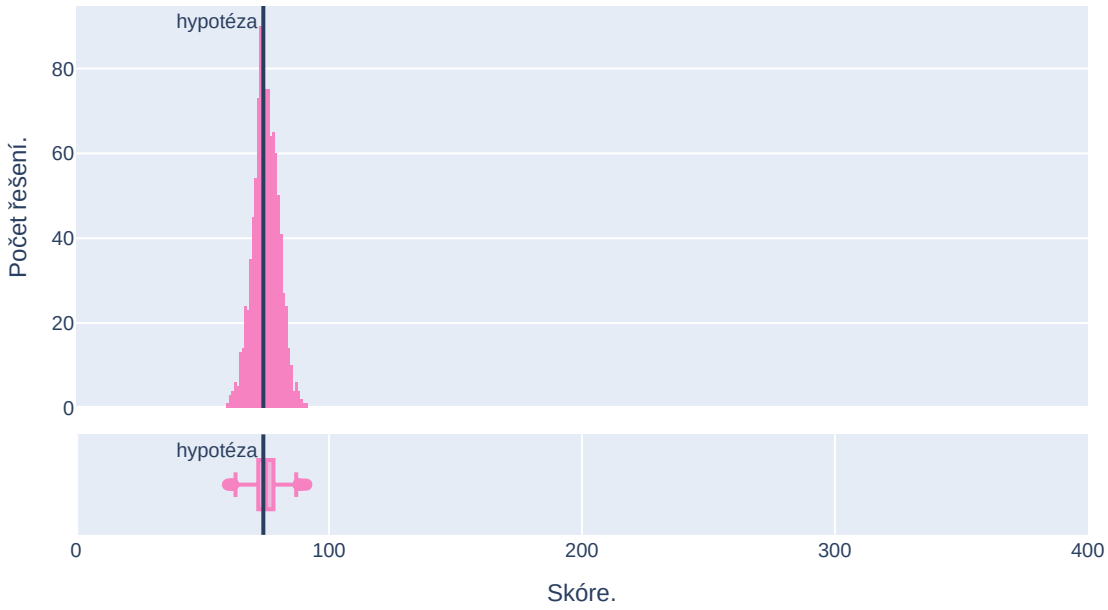
Pokud auta typu t_1 spolu nesousedí, $* \notin N(1) \cup N(o_1)$ a prohození barev aut typu t_1 by zachovalo počet změn, přenastavíme jejich barvy na $*$.

Po návratu ze všech rekurzí zbylé dvojice aut $*$ přebarvíme na dvě různé barvy.

Autoři algoritmu o něm vyslovili domněnku, že $\delta_{\mathbf{rsg}} = \frac{1}{13} \cdot \sqrt{61} - \frac{3}{13} \doteq 0.370$. Na obrázku 2 je zobrazen histogram skóre tohoto algoritmu spuštěného na náhodných vstupech a hodnota z předešlé hypotézy.

2.1 Zobecněné verze a související problémy

Zobecněním binárního paint shopu je obecný paint shop problém, zavedený Eppingem a spol. [5]. Z něj pochází motivace k binární verzi.



Obrázek 2: Graf skóre 1 000 běhů hvězdičkového rekurzivního řešení pro $n = 200$.

Úloha (Obecný paint shop problém): V řadě je m aut n různých typů – nyní však již od každého typu libovolný počet. Dále máme definováno pro každou kombinaci typu a barvy (kterých také může být více než dvě), kolik aut daného typu má být nabarveno na danou barvu. Chceme tedy najít obarvení aut tak, aby počet změn barev v řadě byl co nejmenší.

Binární verze problému je tedy speciální případ obecného paint shop problému, kde od každého typu máme dvě auta, máme jen dvě požadované barvy a navíc platí, že pro každou kombinaci typu a barvy chceme právě jedno auto.

Souvisejícím problémem je dělení náhrdelníku zavedený v Alonově článku [6]:

Úloha (Dělení náhrdelníku): Skupina k loupežníků uloupí náhrdelník skládající se z kn drahokamů t různých typů. Pro každý typ i platí, že náhrdelník obsahuje právě ka_i drahokamů daného typu (pro nějaké celé a_i). Loupežníci si jej chtějí férově rozdělit. Náhrdelník tedy rozdělí na několik navzájem nepřekrývajících se intervalů, jejichž sjednocení je celý náhrdelník. Každý z intervalů pak přiřadí nějakému loupežníkovi tak, aby každý loupežník získat od každého typu i právě a_i drahokamů (což znamená, že všichni budou mít od každého typu stejný počet drahokamů). Chceme minimalizovat počet intervalů, na které je nutné náhrdelník rozdělit.

Nejprve si všimněme, že dělení náhrdelníku je speciálním případem obecného paint shop problému. Auta pro nás budou drahokamy. Barva auta bude označovat, který loupežník získá daný drahokam. Minimalizovat počet intervalů je to stejné jako minimalizovat počet hranic mezi nimi (kterých je o jedna méně něž intervalů). Ve vstupu problému dělení náhrdelníku navíc musí platit, pro každý typ má být stejný počet aut obarvený jednotlivými barvami (a tedy počet aut daného typu musí být násobkem k). Naopak binární paint shop problém je speciálním případem dělení náhrdelníku pro dva lupiče, kde navíc platí, že všechna a_i jsou 1 (tedy od každého typu jsou na náhrdelníku právě dva drahokamy).

3 Semidefinitní programování

V této kapitole si představíme princip semidefinitního programování (dále též SDP), jak jej popisují Gärtner a Matoušek [7], a jeho použití na problém maximálního řezu, z něhož vychází algoritmus na binární paint shop.

Nejprve zavedeme a připomeneme notaci důležitou v této kapitole. Nechť $\mathbb{R}^{n\times m}$ značí množinu n-řádkových m-sloupcových matic složených z reálných čísel. Řádky i sloupce indexujeme od 0, tedy matice $A \in \mathbb{R}^{n\times m}$ obsahuje prvky $A_{i,j}$ pro všechna $0 \leq i < n$ a $0 \leq j < m$. Nechť $\mathrm{SYM}_n = \{X \in \mathbb{R}^{n\times n} \mid x_{i,j} = x_{j,i} \text{ pro všechna } 0 \leq i, j < n\}$ je třída všech symetrických matic a nechť $X \circ Y = \sum_{0\leq i<n} \sum_{0\leq j<m} x_{i,j}y_{i,j}$ značí součet součinu matic po složkách. Nakonec $X \succeq 0$ bude značit skutečnost, že matice X je pozitivně semidefinitní (bude vysvětleno později).

Úloha semidefinitního programování je optimalizační úloha (podobně jako lineární programování) následujícího formátu:

$$\begin{aligned} &\text{maximalizuj} && C \circ X && \\ &\text{za podmínek} && A_i \circ X = b_i && 0 \leq i < m-1 \\ & && X \succeq 0. && \end{aligned}$$

Vstupem programu tedy je velikost matice n, počet podmínek $m \in \mathbb{N}$ a pak pro každou podmínku matice $A_i \in \mathrm{SYM}_n$ a číslo $b_i \in \mathbb{R}$. Výstupem je pak $X \in \mathrm{SYM}_n$ splňující výše uvedené podmínky.

Pro nás bude důležitý následující fakt z lineární algebry:

Fakt: Nechť $X \in \mathrm{SYM}_n$. Následující tvrzení jsou ekvivalentní definice pozitivně semidefinitní matice:

- Všechna vlastní čísla matice X jsou nezáporná.
- Pro každý vektor $\vec{x} \in \mathbb{R}^n$ platí $\vec{x}^{\mathrm{T}} X \vec{x} \geq 0$.
- Existuje matice $Y \in \mathbb{R}^{n\times n}$ taková, že $X = Y^{\mathrm{T}}Y$.

Pro nás bude důležitá zejména třetí podmínka, protože navíc platí, že ze semidefinitní matice X zvládneme zkonstruovat Y v čase $\mathcal{O}(n^3)$ pomocí tzv. Choleského dekompozice.

Navíc pro libovolnou reálnou matici Y je $Y^{\mathrm{T}}Y$ symetrická. Tedy semidefinitní programování můžeme chápat jako optimalizační úlohu na $Y \in \mathbb{R}^{n\times n}$. Pojďme se zamyslet nad tím, co v takovémto pohledu znamenají podmínky a účelová funkce. Matici Y můžeme považovat za n sloupcových vektorů $\overrightarrow{y_0}, \ldots, \overrightarrow{y_{n-1}}$ vedle sebe. Matice Y^{T} pak odpovídá těmto vektorům zapsaných v řádcích pod sebou. Matice $X = Y^{\mathrm{T}}Y$ tedy na pozici i, j obsahuje skalární součin vektorů $\overrightarrow{y_i}$ a $\overrightarrow{y_j}$. Účelová funkce tedy je lineární kombinací skalárních součinů a podmínky odpovídají vynucení rovnosti lineární kombinace skalárních součinů vektorů a konstanty. Speciálně tedy můžeme mít podmínku na délku vektoru: $|\overrightarrow{y_i}|^2 = \overrightarrow{y_i}^{\mathrm{T}}\overrightarrow{y_i} = C$. Semidefinitní programování v tomto tvaru budeme nazývat semidefinitní programování *v dekomponovaném tvaru*.

3.1 Maximální řez

Představíme si pravděpodobnostní aproximační algoritmus založený na semidefinitním programování na následující problém:

Úloha: Nechť $G = (V, E)$ je graf s hranami ohodnocenými nezápornými čísly dle $h : E \to \mathbb{R}_0^+$. Řezem grafu rozumíme rozdělení vrcholů na dvě disjunktní množiny $A \cup B = V$. Hodnotou daného řezu je pak součet cen hran vedoucích mezi A a B. Tedy:

$$H(A, B) = \sum_{\substack{\{u,v\} \in E \\ u \in A, v \in B}} h(u, v)$$

Cílem je maximalizovat hodnotu řezu.

Problém maximálního řezu (resp. rozhodovací verze, kde se ptáme na existenci řezu alespoň dané velikosti) je NP-úplný [8], proto se u něj zkoumají aproximační algoritmy a pravděpodobnostní aproximační algoritmy.

Nejprve si ukážeme triviální pravděpodobnostní 0.5-aproximační algoritmus:

Algoritmus: Každý vrchol uniformně náhodně přiřaď do množiny A nebo B.

Věta: Triviální algoritmus je pravděpodobnostní 0.5-aproximační algoritmus.

Důkaz: Každá hrana bude v řezu s pravděpodobností 1/2 – při umisťování druhého vrcholu dané hrany máme pravděpodobnost 1/2, že ho umístíme do stejné množiny a tedy hrana nebude součásti řezu a pravděpodobnost 1/2 že do opačné a tedy bude součástí řezu. Součet hran v řezu je součtem indikátorů jevů přítomnosti jednotlivých hran v řezu vynásobený jejich hodnotou. Z linearity střední hodnoty tedy střední hodnota součtu vah hran v řezu je 1/2 celkového součtu vah hran, což je alespoň 1/2 optima. *QED*

Povšimněme si, že triviální algoritmus dosáhl poměrně zajímavého aproximačního poměru a to se ani nedíval na vstup.

Předešlý algoritmus lze derandomizovat, jak popisuje Dimitrakakis [9]. Výsledný algoritmus pak vždy najde řešení obsahující alespoň 1/2 hran, tedy alespoň 1/2 optima.

Lepšího aproximačního poměru dosáhneme Goemansovým-Williamsonovým algoritmem, založeným na semidefinitním programování a proto popsaným mimo jiné v již dříve zmíněném úvodu do semidefinitního programování [7].

Naivní implementace je, že si pro každý vrchol u vyrobíme proměnnou x_u, která může nabývat hodnot ± 1, která bude říkat, do jaké množiny máme vrchol umístit. Do účelové funkce pak zakódujeme graf. Účelová funkce bude

$$\sum_{uv \in E} h(u, v) \cdot \left(-\frac{x_u x_v}{2} + \frac{1}{2} \right) = \frac{F}{2} + \frac{1}{2} \sum_{uv \in E} -h(u, v) x_u x_v,$$

kde $F = \sum_{uv \in E} h(u, v)$ značí součet hodnot všech hran. Pro každou hranu tedy přičte $h(u, v)$, pokud x_u a x_v mají různá znaménka, a 0, pokud stejná.

Takovýto optimalizační problém bohužel ani pomocí semidefinitního programování neumíme řešit. Na to, abychom uměli vyjádřit nějaký součin, musíme použít semidefinitního programování v dekomponovaném tvaru. Pak ale místo toho,

abychom měli pro každý vektor jen jednu proměnnou, budeme mít pro každý vrchol celý vektor proměnných a místo součinu proměnných budeme mít skalární součin těchto vektorů. Již ale současně neumíme zařídit, aby tyto vektory byly buď $(1, 0, \ldots, 0)$ nebo $(-1, 0, \ldots, 0)$. Můžeme ale přidat podmínku na to, aby délka každého vektoru byla 1. Tím jsme vyrobili relaxaci výše uvedeného programu, protože za těchto podmínek může vektor nabývat i jiných poloh než $(\pm 1, 0, \ldots, 0)$. Celý algoritmus pak vypadá následovně:

Algoritmus (Goemansův-Williamsonův): Vyřešíme následující semidefinitní program v dekomponovaném tvaru:

$$\begin{aligned} &\text{maximalizuj} && \sum_{uv \in E} -h(u,v)\overrightarrow{y_u}^{\mathrm{T}}\overrightarrow{y_v} \\ &\text{za podmínek} && |y_i| = 1 \qquad 0 \leq i < |V| \end{aligned}$$

Dále uniformně náhodně zvolíme jednotkový vektor vektor $z \in \mathbb{R}^n$ a do jedné množiny vybereme právě ty vrcholy v, pro které $y_v^{\mathrm{T}} z \geq 0$.

Semidefinitní program v algoritmu si lze představit jako umísťování n bodů na $n-1$ dimenzionální sféru v $\mathbb{R}^n$. Účelová funkce se snaží umístit body vrcholů spojených hranou co nejdále od sebe.

V případě, kdy jsme povolovali pouze hodnoty ± 1, bylo jasné, které vrcholy patří do které množiny. Nyní však výstupem optimalizace jsou vektory a tedy přiřazení množinám není tak jednoznačné. Rádi bychom umístili od sebe vzdálené body do různých množin. Na to můžeme rovnoměrně náhodně zvolit nadrovinu procházející počátkem a rozdělit body do množin podle toho, do které z polorovin určených danou nadrovinou patří.[1]

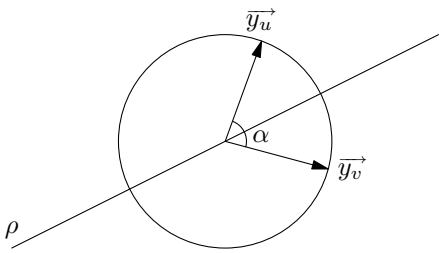
To nám zaručí, že dvojice bodů daleko od sebe budou mít velkou pravděpodobnost toho, že budou v různých poloprostorech. Semidefinitní programování se tedy snaží dostat dvojice bodů odpovídající vrcholům spojených hranou daleko od sebe a podle toho se zvyšuje pravděpodobnost toho, že vrcholy budou v jiných množinách a tedy hrana bude v řezu.

Výběr nadroviny a rozdělování bodů jde implementovat pomocí výběru vektoru $\vec{z}$ kolmého na nadrovinu. Pro každý bod pak stačí spočítat skalární součin se $\vec{z}$ a podle znaménka víme, do kterého poloprostoru patří. Pokud vybereme uniformně náhodný jednotkový vektor, tak jsme uniformně náhodně vybrali nadrovinu. Jednotkový vektor můžeme generovat tak, že vygenerujeme náhodný vektor nezávisle po složkách z normálního rozdělení, a pak ho znormujeme. Povšimněme si, že normalizace ani není potřeba, protože to na znaménku součinů nic nemění.

Nyní pojďme precizněji spočítat pravděpodobnost toho, že se dvojice bodů (vektorů) $\overrightarrow{y_u}$, $\overrightarrow{y_v}$ rozdělí náhodnou nadrovinou ρ v závislosti na hodnotě skalárního součinu mezi nimi. Můžeme se podívat na rovinu, ve které leží počátek a oba vektory $\overrightarrow{y_u}$, $\overrightarrow{y_v}$ (viz obrázek 3). Průnik náhodné nadroviny s touto rovinou tvoří rovnoměrně náhodně vybranou přímku procházející počátkem.

Zajímá nás tedy, jaká je pravděpodobnost toho, že jedno z ramen nadroviny bude uvnitř konvexního úhlu mezi $\overrightarrow{y_u}$ a $\overrightarrow{y_v}$. Velikost tohoto úhlu označme α. Víme,

[1] Předpokládejme, že žádný z bodů neleží na nadrovině. Pravděpodobnost náležení nadrovině je 0. Pokud se tak stane, je vcelku jedno, do jaké množiny ho vložíme.

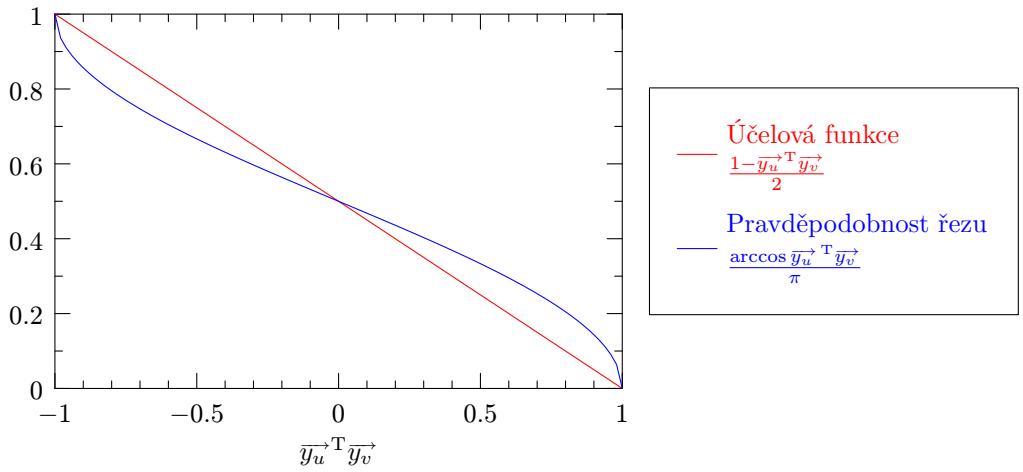


Obrázek 3: Znázornění řezu rovinou obsahující $\overrightarrow{y_u}$ i $\overrightarrow{y_v}$.

že $\overrightarrow{y_u}^{\mathrm{T}}\overrightarrow{y_v} = 1 \cdot 1 \cdot \cos\alpha$. Náhodnou přímku můžeme vygenerovat jako náhodný úhel β mezi 0 a 2π s tím, že přímka pak povede směrem β a $(\beta+\pi)$ mod 2π. Nikdy však uvnitř konvexního úhlu nebudou obě ramena přímky, tedy pravděpodobnost, že alespoň jedno bude uvnitř a tedy body budou oddělené je:

$$\frac{2 \cdot \alpha}{2\pi} = \frac{\alpha}{\pi} = \frac{\arccos \overrightarrow{y_u}^{\mathrm{T}}\overrightarrow{y_v}}{\pi}$$

U každé hrany uv optimalizujeme $-h(u,v)\overrightarrow{y_u}^{\mathrm{T}}\overrightarrow{y_v}$, což je ekvivalentní optimalizování $h(u,v) \cdot \left(\frac{1}{2} - \frac{\overrightarrow{y_u}^{\mathrm{T}}\overrightarrow{y_v}}{2}\right)$.



Obrázek 4: Graf funkcí pravděpodobnosti řezu a účelové funkce.

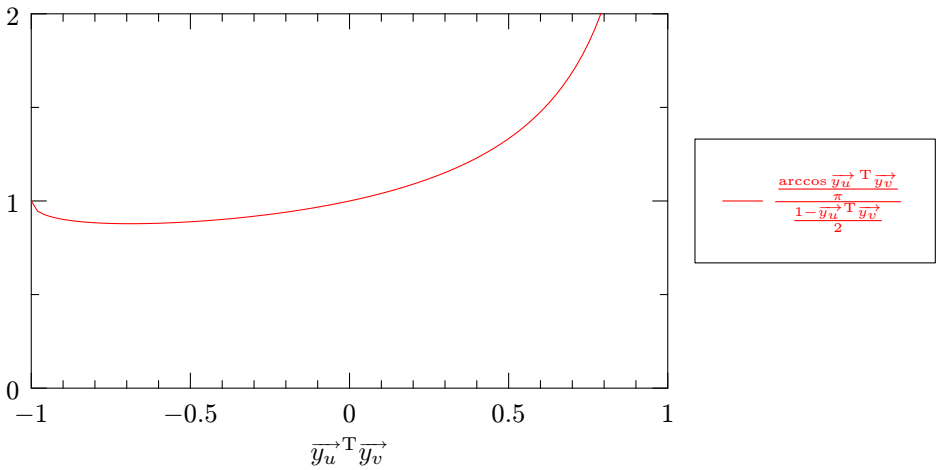
Nyní se podívejme na poměr mezi pravděpodobností toho, že hrana bude v řezu, a účelovou funkcí před vynásobením hodnotou hrany jakožto funkci proměnné $x = \overrightarrow{y_u}^{\mathrm{T}}\overrightarrow{y_v}$ v intervalu $[-1, 1]$:

$$\frac{\arccos \overrightarrow{y_u}^{\mathrm{T}}\overrightarrow{y_v}}{\pi} / \frac{1 - \overrightarrow{y_u}^{\mathrm{T}}\overrightarrow{y_v}}{2} = \frac{\arccos x}{\pi} / \frac{1-x}{2}$$

Fakt: Pro každé $x \in [-1, 1]$ platí:

$$\frac{\arccos x}{\pi} / \frac{1-x}{2} > 0.8785$$

Označme tuto hodnotu $c = 0.8785$.



Obrázek 5: Graf poměru pravděpodobnosti řezu a účelové funkce.

Tedy když je účelová funkce dané hrany $h(u,v)r$, tak pravděpodobnost výběru hrany do řezu bude alespoň cr. Sečtením přes všechny hrany (z linearity středních hodnot) tedy získáme, že střední hodnota součtu hran v řešení je alespoň cR, kde $R = \sum_{uv\in E} h(u,v) \cdot \left(\frac{1}{2} - \frac{\overrightarrow{y_u}^{\mathrm{T}}\overrightarrow{y_v}}{2}\right)$, tedy jedno z ekvivalentních vyjádření účelové funkce.

Pokud tedy najdeme dobré řešení semidefinitního programu, ve střední hodnotě pak najdeme i dobré řešení maximálního řezu. Dále však musí platit, že optimální řešení semidefinitního programu je alespoň součet hran optimálního řešení maximálního řezu, když za účelovou funkci považujeme $\sum_{uv\in E} h(u,v) \cdot \left(\frac{1}{2} - \frac{\overrightarrow{y_u}^{\mathrm{T}}\overrightarrow{y_v}}{2}\right)$. Jedno z možných řešení totiž snadno sestrojíme z maximálního řezu: Vrcholům jedné množiny přidělíme vektory $(1,0,\ldots,0)$ a druhé $(-1,0,\ldots,0)$. Účelová funkce pak přičte $h(u,v)$ za každou hranu v řezu a 0 jinak.

Bohužel na rozdíl od lineárního programování, u semidefinitního programování nejsme schopní v polynomiálním čase najít optimální řešení. Naštěstí však platí, že v polynomiálním čase za určitých podmínek jsme schopní se mu libovolně přiblížit. Přesněji řečeno, pro každé $\varepsilon > 0$ jsme schopní najít řešení s účelovou funkcí nejdále ε od optima. Ovšem musí platit, že všechna přípustná řešení jsou dostatečně malá. Přesněji řečeno, musí platit, že Frobeniova norma všech přípustných matic je omezena konstantou s polynomiální délkou zápisu. Tohoto výsledku jde dosáhnout pomocí elipsoidové metody, která má nejlepší teoretické výsledky. V praxi se však často používají jiné algoritmy. Všechny naše programy budou splňovat podmínky na řešení, protože každá z hodnot hledané matice nutně bude ležet v intervalu $[-1,1]$.

Věta: Goemansův-Williamsonův algoritmus je pravděpodobnostní 0.878-aproximační algoritmus.

Důkaz: Využitím předešlých pozorování a elipsoidové metody získáme (MC značí max cut, tedy maximální řez):

$$\mathbb{E}[\text{Řešení MC}] \geq c\cdot \text{Řešení SDP} \geq c(1-\varepsilon)\cdot \text{Optim. SDP} \geq c(1-\varepsilon)\cdot \text{Optim. MC}.$$

Pro dostatečně malé ε se tedy jedná o pravděpodobnostní 0.878-aproximační algoritmus. *QED*

4 Řešení semidefinitním programováním

Řešení vychází z Goemansova-Williamsonova algoritmu na maximální řez. Pro každé auto nám bude semidefinitní program umísťovat bod (někdy také chápán jako vektor) na jednotkovou sféru. Poté náhodně rozřízneme nadrovinou sféru na dvě poloviny a podle toho, do které poloviny auto patří, zvolíme jeho barvu.

Rozdílné barvy aut i, j stejného typu zařídíme tak, že vynutíme $\overrightarrow{y_i} = -\overrightarrow{y_j}$. Na to nám stačí jediná podmínka – říct, že jejich skalární součin je -1.

Účelovou funkcí pak řekneme, že sousední auta mají preferovat stejnou barvu, tedy jejich body na sféře mají být blízko sebe, což znamená, že skalární součin má být co největší.

Algoritmus (Řešení pomocí semidefinitního programování, sdp):
Vyřešíme následující semidefinitní program v dekomponovaném tvaru:

$$\begin{aligned} &\text{maximalizuj} && \sum_{0 \le i < 2n-1} \overrightarrow{y_i}^{\mathrm{T}} \overrightarrow{y_{i+1}} \\ &\text{za podmínek} && \overrightarrow{y_{a_{i,0}}} = -\overrightarrow{y_{a_{i,1}}} && 0 \le i < n \\ & && |y_i| = 1 && 0 \le i < 2n \end{aligned}$$

Dále uniformně náhodně zvolíme jednotkový vektor $z \in \mathbb{R}^{2n}$ a červeně obarvíme právě ty auta i, pro která $y_i^{\mathrm{T}} z \ge 0$. Náhodných výběrů vektoru je možné provést vícero a pak vybrat nejlepší nalezené řešení.

Pro každou dvojici do účelové funkce přičteme číslo mezi -1 a 1, kde 1 značí, že vektory jsou stejné, a -1, že jsou protilehlé. Účelovou funkci můžeme ekvivalentně zapsat jako

$$\text{minimalizuj} \qquad \sum_{0 \le i < 2n-1} \frac{1}{2} - \frac{\overrightarrow{y_i}^{\mathrm{T}} \overrightarrow{y_{i+1}}}{2} = \frac{2n-1}{2} - \frac{1}{2} \sum_{0 \le i < 2n-1} \overrightarrow{y_i}^{\mathrm{T}} \overrightarrow{y_{i+1}}$$

Nyní tedy pro každou dvojici sousedních aut přičítáme číslo mezi 0 a 1, kde 0 přičteme pro stejné vektory, mezi nimiž nikdy nebude změna barvy a 1 přičteme v případě opačných vektorů, mezi nimiž se nutně barva změní. Účelovou funkci se snažíme minimalizovat.

Věta: Optimum semidefinitního programu, který minimalizuje účelovou funkci $\frac{2n-1}{2} - \frac{1}{2} \sum_{0 \le i < 2n-1} \overrightarrow{y_i}^{\mathrm{T}} \overrightarrow{y_{i+1}}$ je menší rovno počtu změn v optimálním obarvení aut.

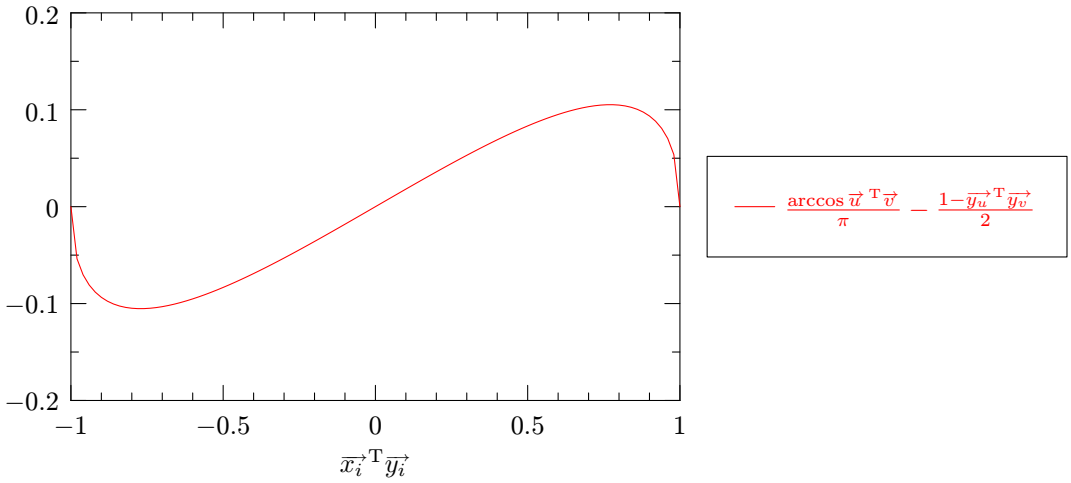
Důkaz: Podíváme se na optimální obarvení aut a každému červenému autu přiřadíme vektor $(1, 0, \ldots, 0)$ a modrému $(-1, 0, \ldots, 0)$. Toto je validní řešení semidefinitního programu. Účelová funkce za každou změnu barev přičte 1 a zbytek součtu je 0, takže její hodnota je přesně počet změn barev. *QED*

Nyní by se hodilo odhadnout střední hodnotu počtu změn barev stejně jako u maximálního řezu. Jeho důkaz vycházel z pozorování ohledně poměru pravděpodobností výběru a účelové funkce, přesněji řečeno z toho, že tento poměr umíme zdola odhadnout. Ovšem v našem případě místo maximalizace děláme minimalizaci účelové funkce a pravděpodobnosti. Tedy abychom mohli tvrdit, že pravděpodobnost je menší než nějaký násobek účelové funkce, potřebovali bychom horní

odhad poměru. Ovšem tento poměr je neomezený (viz obrázky 4 a 5). V okolí $x = 1$ jsou obě funkce poblíž 0, ovšem pravděpodobnost se k nule blíží mnohem strměji. Tedy pro dvojici vektorů poblíž sebe je skalární součin skoro 1, ovšem pravděpodobnost oddělení je libovolně krát větší než vzdálenost součinu od jedné.

O binárním paint shop problému je navíc známé, že je za předpokladu Unique game conjecture a P $\neq$ NP je polynomiálně neaproximovatelný s konstantním faktorem [2], takže nemožnost výše uvedeného postupu by nás ani neměla zaskočit, protože v případě, že by šlo udělat odhad tímto způsobem, získali bychom pravděpodobnostní aproximační algoritmus.

Místo toho se podíváme na rozdíl pravděpodobnosti a účelové funkce.



Obrázek 6: Graf rozdílu pravděpodobnosti řezu a účelové funkce.

Lemma: Pro každé $x \in [-1, 1]$ platí:

$$-0.1053 \leq \frac{\arccos x}{\pi} - \left(\frac{1}{2} - \frac{x}{2}\right) \leq 0.1053$$

Důkaz: Jedná se o spojitou funkci na kompaktním intervalu, takže maximum a minimum může nabývat jen v krajních bodech a v místech s nulovou derivací. V obou krajních bodech je však její hodnota 0, zbývá tedy prověřit nulové derivace. Zderivováním získáme:

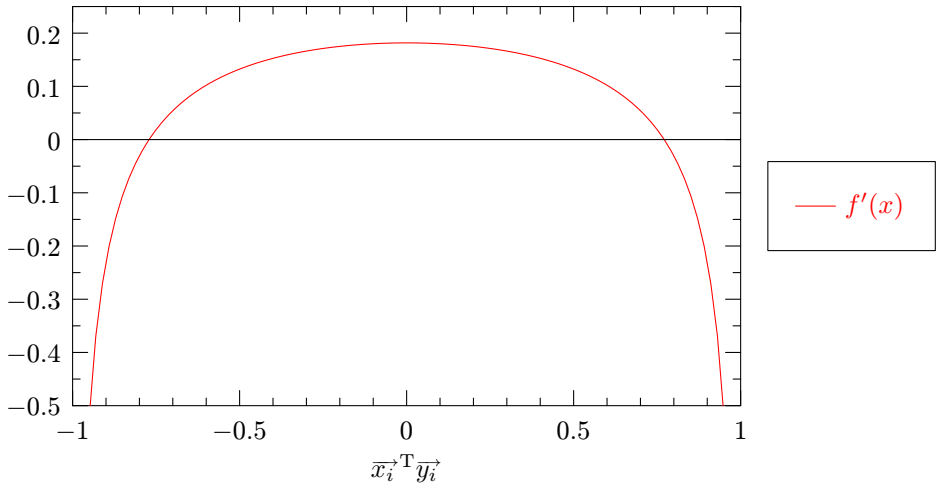
$$f'(x) = -\frac{1}{\pi\sqrt{-x^2+1}} + \frac{1}{2},$$

což má dva kořeny $x = \pm\sqrt{\pi^2 - 4}/\pi$, a hodnota h v nich je přibližně ± 0.105256. *QED*

Označme konstantu z předchozího lemmatu jako $d = 0.1053$.

Věta: Střední hodnota (vzhledem k náhodným bitům generovaným algoritmem) počtu změn v **sdp** řešení je nejvýše o $0.212n$ horší než optimum.

Důkaz: Když účelová funkce pro sousední dvojici aut je r, tak pravděpodobnost změny barev je nejvýše $d + r$. Sečtením přes všechny hrany (z linearity středních



Obrázek 7: Derivace funkce f.

hodnot) tedy získáme, že střední hodnota součtu změn je nejvýše $d(2n-1)+R$, kde $R = \sum_i^{2n-1} \frac{1}{2} - \frac{\overrightarrow{y_i}^{\mathrm{T}}\overrightarrow{y_{i+1}}}{2}$, tedy jedno z ekvivalentních vyjádření účelové funkce. Celkově tedy víme:

$$\mathbb{E}[\gamma_{\mathbf{sdp}}(\alpha)] \leq 2dn + \text{Řeš. SDP} \leq 2dn + \text{Opt. SDP} + \varepsilon \leq 2dn + \gamma(\alpha) + \varepsilon$$

Pro vhodně zvolené ε jsme tedy dokázali požadovanou vzdálenost od optima. *QED*

Na rozdíl od jiných řešení binárního paint shop problému, zde se jedná o střední hodnotu přes náhodné rozhodnutí algoritmu, nikoliv přes náhodnou distribuci na vstupech. Tedy když spustíme algoritmus na konkrétní instanci, dostaneme řešení a také dolní odhad na optimalní řešení dané instance (ten jednak můžeme odvodit z hodnoty řešení ale také přímo z účelové funkce semidefinitního programu).

4.1 Implementační zjednodušení

Předchozí semidefinitni program ještě lze nepatrně zjednodušit se zachováním všech potřebných vlastností. Místo toho, abychom měli vektor pro každé auto, uděláme si jen vektor pro každou dvojici aut stejného typu. Ten bude odpovídat řekněme prvnímu autu z dvojice. Všimneme si, že účelovou funkci zvládneme stále vyjádřit. Kdykoliv jsme používali vektor druhého auta, stačí použít mínus vektor prvního auta. Tedy stačí některé členy jednou až dvakrát vynásobit hodnotou -1.

Výsledný program vypadá takto:

$$\begin{aligned} &\text{maximalizuj} && \sum_{0 \leq i < 2n-1} \overrightarrow{y_{t_i}}^{\mathrm{T}} \overrightarrow{y_{t_{i+1}}} P_i P_{i+1} \\ &\text{za podmínek} && |y_i| = 1 && 0 \leq i < n \end{aligned}$$

Výhodou je, že je potřeba poloviční počet proměnných, matice je čtvrtinová a odebrali jsme n podmínek, proto v implementační části práce využijeme této formy programu.

5 Měření řešení BPS

Součástí práce je implementace algoritmů řešících Binární paint shop problém. Každý z nich byl následně spuštěn pro různé velikosti, pokaždé na 1 000 nezávisle náhodně vybraných vstupech s počtem typů aut 10, 20, 50, 100, 200, 400, 566, 800, 1131, 1600, 2263 a 3200.[2]Jedna z implementací **sdp** – pomocí sage vyžaduje velké množství paměti a proto byla spuštěna jen na vstupech do velikosti 566. Celý test běžel několik dní na čtveřici stojů, na každém dva programy současně, každý z nich běžel jednovlákonvě a využíval nejvýše 16 GB operační paměti. U **sdp** řešení bylo použito 10 náhodných řezů a vždy byl vybraný nejlepší z nich. Gitový repozitář s implementací i naměřenými daty je k dispozici na *https://gitlab.kam.mff.cuni.cz/jirikalvoda/binary-paint-shop-problem*. Na následujících stranách jsou zpracovaná různá naměřená data.

Pro reprodukovatelnost byly vstupy generovány deterministicky ze seedů. Se stejným seedem se tedy vždy vygeneruje stejný vstup a ideálně i stejné řešení. Pro různé algoritmy a velikosti vstupů byly použité různé seedy.

5.1 Praktické řešení semidef. programů

U většiny algoritmů byla implementace poměrně přímočará. Ovšem u řešení semidefinitního programování je většina složitosti algoritmu schovaná právě v řešení semidefinitních programů, což už svojí složitostí nepatří mezi algoritmy, které bychom chtěli (re)implementovat. Proto je žádoucí se spolehnout na funkčnost již existujících implementací. Bohužel kvalita dostupných řešičů semidefinitních programů je poměrně nízká.

5.1.1 SDPA-C

Jedním z použitých programů byl SDPA-C [10]. Jedná se o knihovnu v C++ založenou na metodě vnitřních bodů v primárním a duálním problému. Dle autorů projektu [10]:

> SDPA (SemiDefinite Programming Algorithm) is one of the most efficient and stable software packages for solving SDPs based on the primal-dual interior-point method. It fully exploits the sparsity of given problems.

Pro účely práce byla použita verze SDPA-C ze dne 2023-06-21 (soubor `sdpa-c.7.3.8.src.20180125.tar.gz`). Kompilace je poměrně komplikovaná. Jednak vyžaduje velké množství nainstalovaného software (například překladač Fortranu), ale také poskytnutý `Makefile` není plně kompatibilní s moderními verzemi překladačů.

Bohužel i po úspěšné kompilaci stále není vyhráno. Dodaný program přeložený aktuálním kompilátorem bohužel nefunguje.

Při spuštění i na malých vstupech program rychle spadne s chybou Segmentation Fault ve funkci `Newton::compute_bMatgVec_dense_threads_SDP`.

Při následujícím ladění programu bylo zjištěno, že se program zacyklí v dané funkci ve smyčce, při které inkrementuje hodnotu proměnné `Column_Number` až

[2] Hodnoty, co nejsou hezká čísla jsou zhruba $\sqrt{2}$-násobky předešlé hodnoty, tedy v polovině mezi okolními hodnotami na logaritmické stupnici.

do doby, kdy přeteče, což už přímo vede k pádu programu. Pohledem na kód to vypadá, že toto by se nemělo dít. Program už mnohokrát měl ukončit smyčku a doběhnout na konec funkce, ovšem po přidání dostatečného počtu debugovacích výpisů je vidět, že program cyklus opustí, provede příkazy mezi cyklem a koncem funkce a pak se zase objeví uprostřed cyklu.

Problém byl v tom, že funkce vracející `void*` dle normy [11] bodu 6.6.3 musí skončit pomocí `return` a nemůže jen tak opustit funkci dojitím na její konec. Pokud se tak stane, jedná se o nedefinované chování, nikoliv jen o nedefinovanou hodnotu navrácenou z funkce. Kdyby se jednalo jen o nedefinovanou hodnotu, tak se nic zásadního neděje. Funkce může vrátit cokoliv, ale to je nám jedno, protože výsledek se ignoruje.

Ovšem jelikož se jedná o nedefinované chování, překladač může program přeložit tak, že v takovémto případě udělá prakticky cokoliv. A v tomto případě optimalizace překladače přeložili program tak, že v případě opuštění funkce se vykonávání programu vrátí zpět dovnitř a pokračuje se v iterování smyčky. To je důsledkem toho, jakým způsobem se snaží překladač optimalizovat kód. Překladače nemusí dodržovat naprogramovanou strukturu programu. Tedy není problém přesunout část za smyčkou do ní na místo, kde se smyčka opouští. Když překladač ví, že v korektním programu by se tudy nemělo opouštět funkci, nemusí se překladač obtěžovat umísťováním explicitního návratu z funkce na dané místo.

Naštěstí oprava tohoto problému je přímočará – stačí do takovýchto funkcí explicitně dopsat `return NULL;`.

Toto bohužel není jediný problém s SDPA-C. Na některých vstupech knihovna dojde do stavu, kdy nemá semidefinitní matici a výpočet spadne s následující chybou:

```
CHOLMOD warning: matrix not positive definite.
file: ../Supernodal/t_cholmod_super_numeric.c line: 911
getMinEigenValue:: cannot decomposition ::
line 193 in sdpa_linear.cpp
```

Tedy během výpočtu program něco spočte špatně a jako mezivýsledek mu vyjde matice, co není pozitivně semidefinitní, se kterou již dále nejde pokračovat. Příčinu této chyby se bohužel nepodařilo odhalit, protože na rozdíl od předchozího problému vůbec není jasné, kde začít hledat. Detekce nesemidefinitnosti matice totiž může být v úplně jiné části kódu, než kde nastane chyba.

Problém se projevuje například na vstupu s $n = 5$ a seedem 19.

Další problém (nebo ten stejný) se projevuje zejména u větších vstupů. Uprostřed algoritmu se také objeví chybová hláška o nesemidefinitnosti matice. Následně program udělá mnoho iterací s nekonečnou účelovou funkcí, dokud se nepřekročí maximální počet iterací, a pak je vrácena matice ze samých $\pm\infty$, což dokonce porušuje zadané podmínky. Toto se stává například na seedu 18 pro $n = 150$.

Vzhledem k výše uvedeným problémům dává smysl se porozhlédnout po alternativách.

5.1.2 Sage

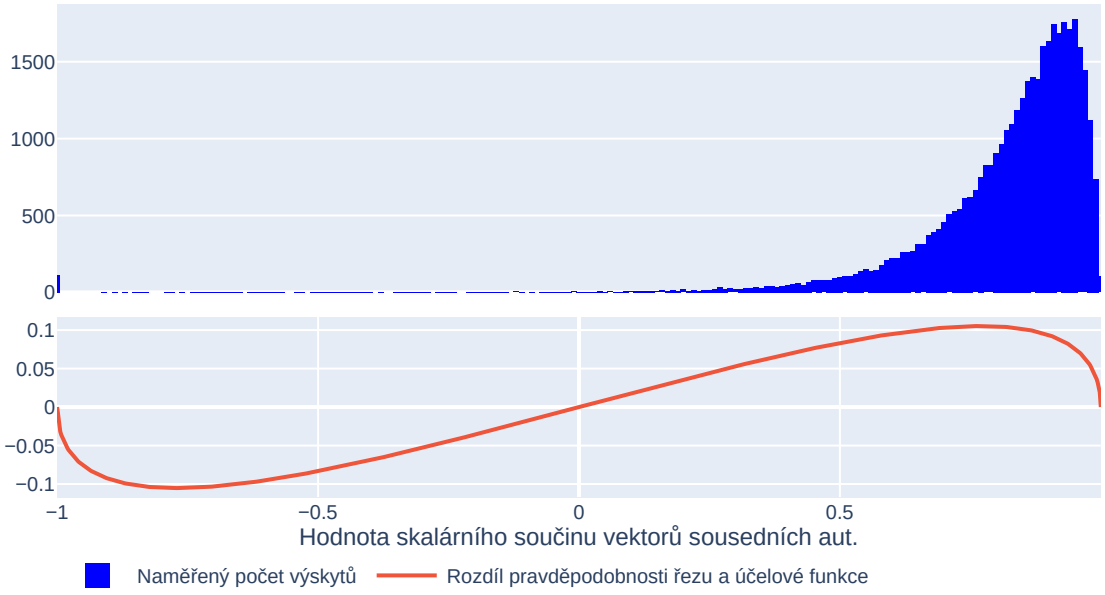
SageMath je dle oficiálních stránek [12] open-source software pro matematické výpočty založený na rozšířené syntaxi pythonu, takže je poměrně snadné ho využívat i jako programovací jazyk. Pro účely práce byla použita verze sage 10.1 ze dne 2023-08-20. Sage mimo jiné obsahuje rozhraní pro řešení semidefinitních programů. Rozhraní je navrženo tak, aby bylo schopné pracovat s více různými backendy pro řešení semidefinitních programů. Bohužel z dokumentace [13] nebylo zřejmé, jaké backendy sage podporuje. Při zavolání `default_sdp_solver("")` se objeví chybová hláška, která tvrdí, že backend by mělo jít nastavit na `CVXOPT` (což je výchozí hodnota) nebo `Matrix`, případně uživatelem dodanou třídu. Ovšem při přepnutí na `Matrix` spuštění řešení spadne na `NotImplementedError`, tedy zdá se, že ani tento backend není implementovaný (alespoň ve verzi, co mám k dispozici) a tedy jediná možnost je `CVXOPT`.

Drobným problémem je, že dle dokumentace [13] rozhraní sage vyžaduje formulování vstupu jako primárního semidefinitního programu, ovšem náš algoritmus je založený na řešení duálního semidefinitního programu. Mezi nimi však lze snadno přecházet pomocí triviální úpravy, ovšem výsledný kód pak vypadá velice neintuitivně.

5.2 Hodnoty skalárních součinů

V odhadu střední hodnoty počtu změn barev v **sdp** řešení jsme využívali toho, že funkce rozdílu pravděpodobnosti řezu a účelové funkce (funkce f) je nejvýše $d = 0.1053$. Ovšem tato funkce nenabývá takto vysokých hodnot zdaleka všude, na polovině definičního oboru je dokonce záporná (viz obrázek 6). Kdyby se tedy alespoň část optimalizovaných skalárních součinů vyskytovala mimo oblast, kde f nabývá vysokých hodnot, šel by odhad zlepšit.

Bohužel dle naměřených dat to vypadá, že hodnoty skalárních součinů se koncentrují pouze poblíž maxima f. Viz obrázek 8. Tento výsledek intuitivně dává smysl, protože jenom takto může semidefinitní program dosáhnout lepší účelové funkce než počet změn barev v optimálním řešení. Ovšem náš jediný dolní odhad je hodnota semidefinitního programu a tedy nejsme schopní více přiblížit dolní a horní odhad k sobě.



Obrázek 8: Distribuce skalárních součinů
100 běhů algoritmu pro $n = 200$.

Z grafu je vidět, že skoro nikdy semidefinitní program neumístí sousední body do protilehlých částí. Drobnou výjimku tvoří hodnota skalárního součinu okolo -1, kterých semidefinitní program za 100 běhů vyrobil zhruba 100. K tomuto mohl být donucen existencí dvojic aut stejného typu hned vedle sebe, kdy nemá jinou možnost než mezi nimi mít skalární součin -1.

Lemma: Střední hodnota počtu sousedních aut stejného typu je 1.

Důkaz: Pro každý typ auta máme $\binom{2n}{2} = \frac{2n(2n-1)}{2} = n(2n-1)$ možných pozic, kde se mohou nacházet a z toho $2n-1$ z nich jsou vedle sebe. Střední hodnota indikátoru souslednosti aut daného typu je tedy $\frac{2n-1}{n(2n-1)} = \frac{1}{n}$. Z linearity střední hodnoty tedy počet sousedících aut stejného typu je $n/n = 1$. *QED*

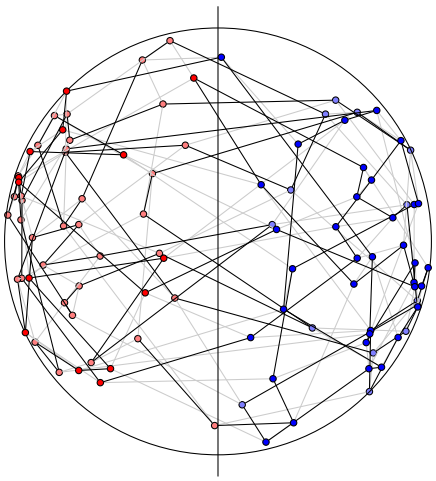
Ve 100 vstupech by tedy sousedních aut stejného typu mělo být ve střední hodnotě zhruba 100, což z části vysvětluje pozorovanou anomálii. Není však vyloučeno, že hodnoty skalárního součinu -1 nebo blízké program dosáhne i jinak.

5.3 Dimenze SDP

Semidefinitní program rozmísťuje vektory do $n-1$ dimenzionální sféry. Z naměřených dat však vychází, že semidefinitní program většinou generuje řešení, které má mnohem menší dimenzi. Přesněji řečeno, pro řešení často platí, že všechny vektory v něm mají několik prvních souřadnic velké hodnoty a ve zbylých souřadnicích mají hodnoty blízké nule. Tedy kdybychom vektory promítli na méně dimenzionální sféru (vzali místo nich nejbližší bod na ní), tak se účelová funkce moc nezmění.

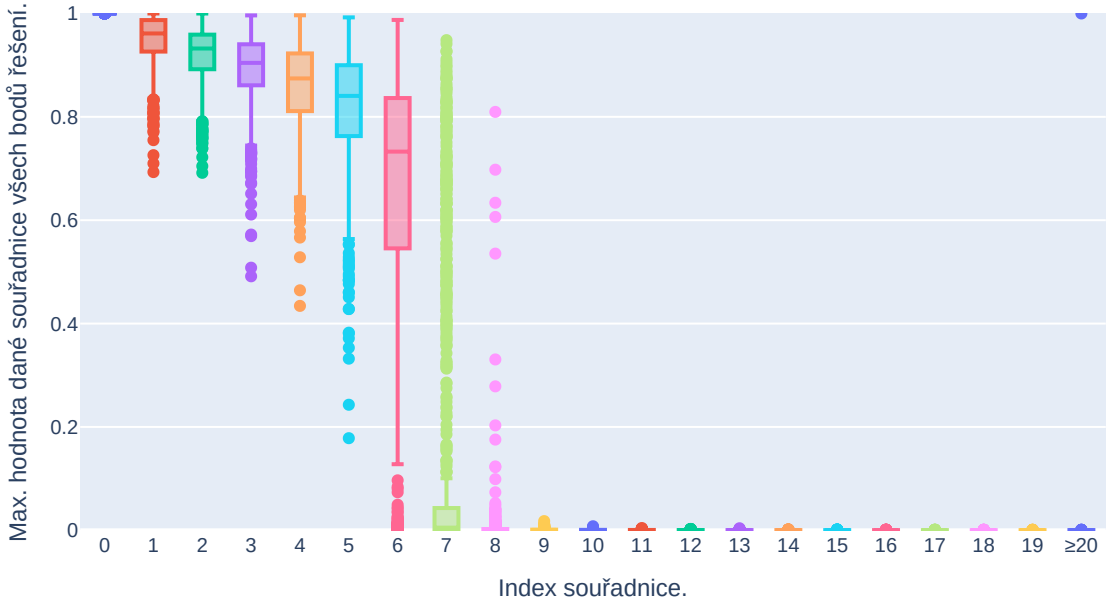
Nevíme o žádné hypotéze, proč program generuje řešení s malou dimenzí. Taktéž není jasné, jestli či jak by toho šlo využit pro získání lepšího řešení či dolního odhadu. Z naměřených dat zobrazených na obrázcích 10 až 17 na následujících stranách však vychází, že průměrná dimenze je mnohem menší než n, ovšem nejspíše není omezena žádnou konstantou, tedy s rostoucím n také roste, ale mnohem pomaleji. Hypotéza zní, že průměrná dimenze leží v $\Theta(\log n)$.

Malá dimenze má ale zásadní výhodu pro vizualizaci řešení, protože trojrozměrný prostor (tedy sféru dimenze 2) si zvládne člověk snadno představit. Z čehož plyne, že zhruba polovinu řešení pro $n = 50$ jsme schopni zakreslit jen s drobným zkreslením, tak, jak je ukázáno na obrázku 9.

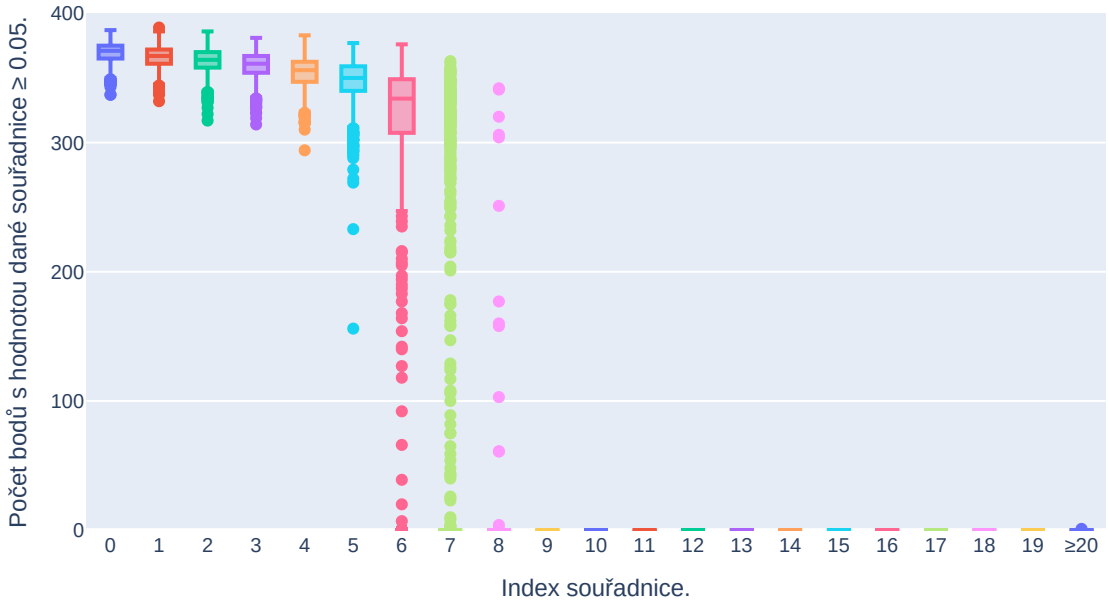


Světlá barva bodu značí bod na zadní straně sféry. Černé čáry spojují sousední auta a šedé jsou jim středově symetrické (protože dvojice aut, jejichž druhá auta stejného typu jsou vedle sebe, je také přitahována k sobě).

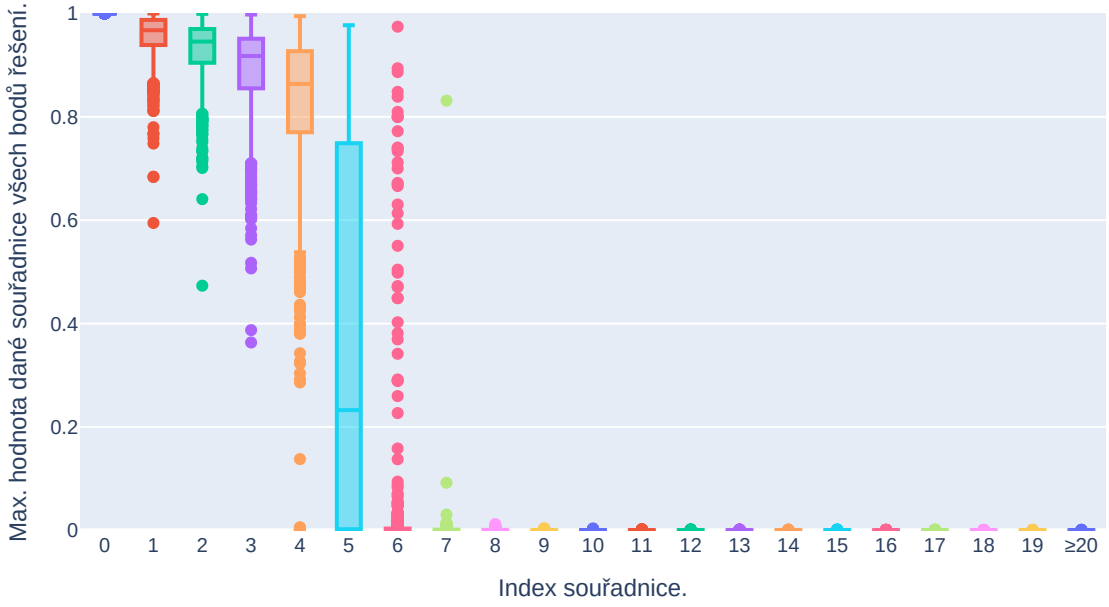
Obrázek 9: Vizualizace jednoho z řešení pro $n = 50$, které se vejde do 3D.



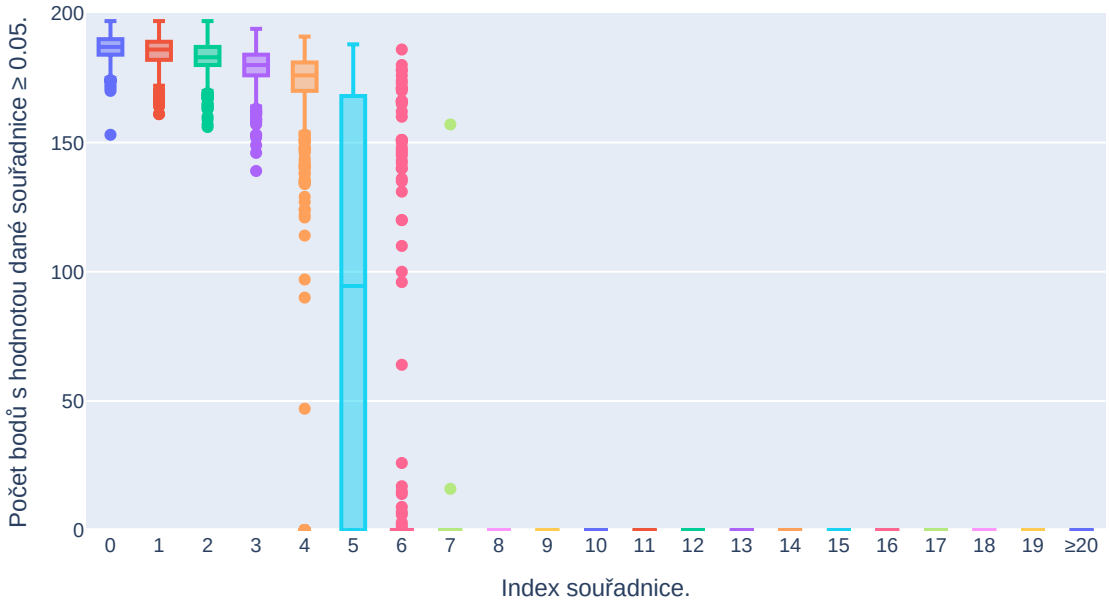
Obrázek 10: Maximální hodnota v dané dimenzi pro $n = 400$.



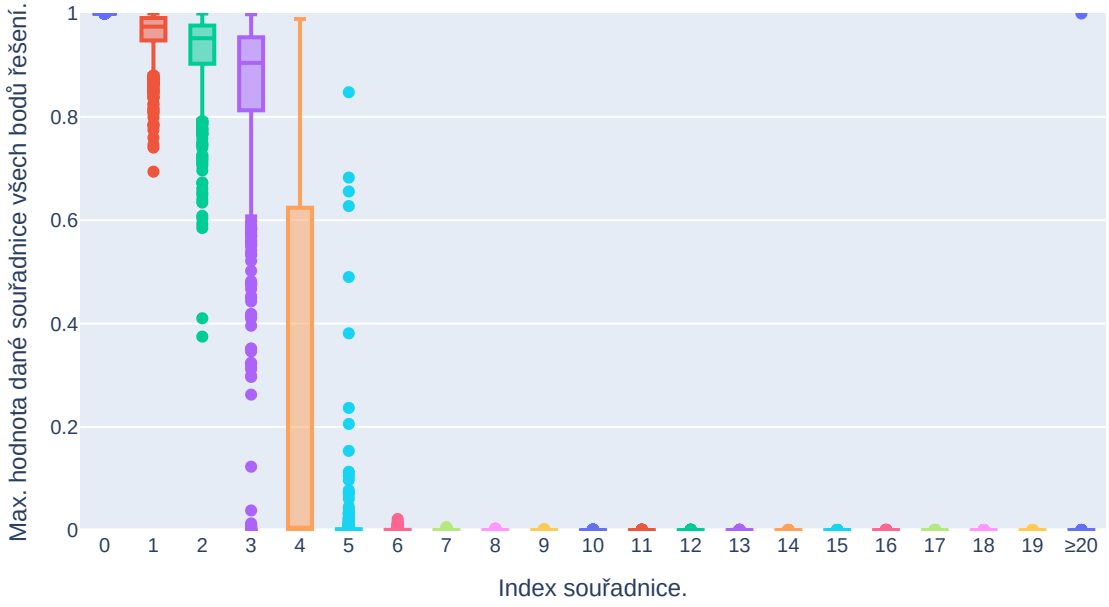
Obrázek 11: Počet vektorů s danou souřadnicí větší než 0.05 pro $n = 400$.



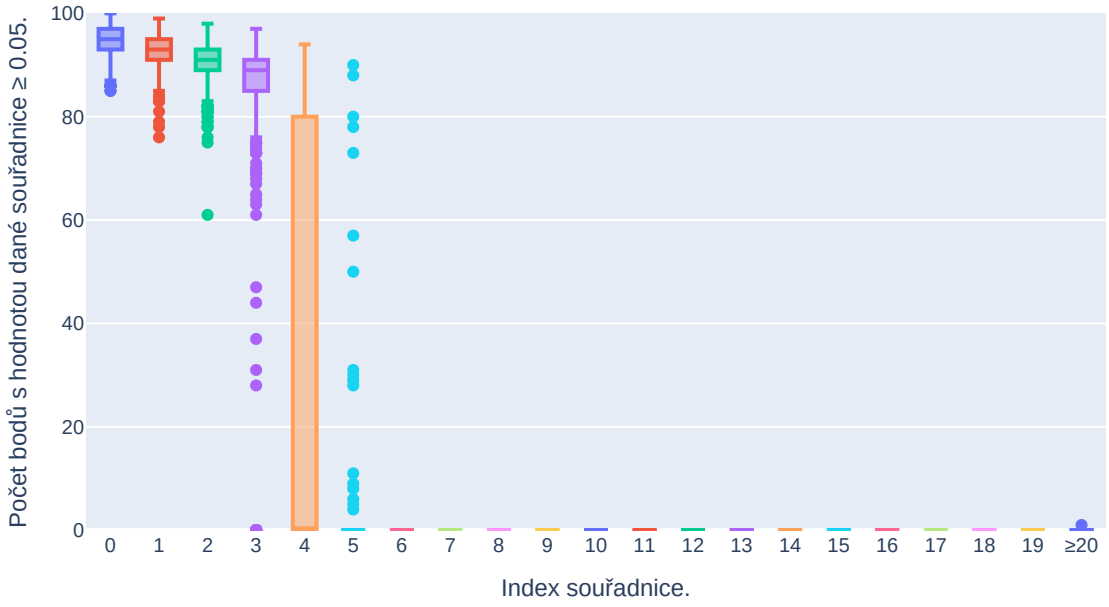
Obrázek 12: Maximální hodnota v dané dimenzi pro $n = 200$.



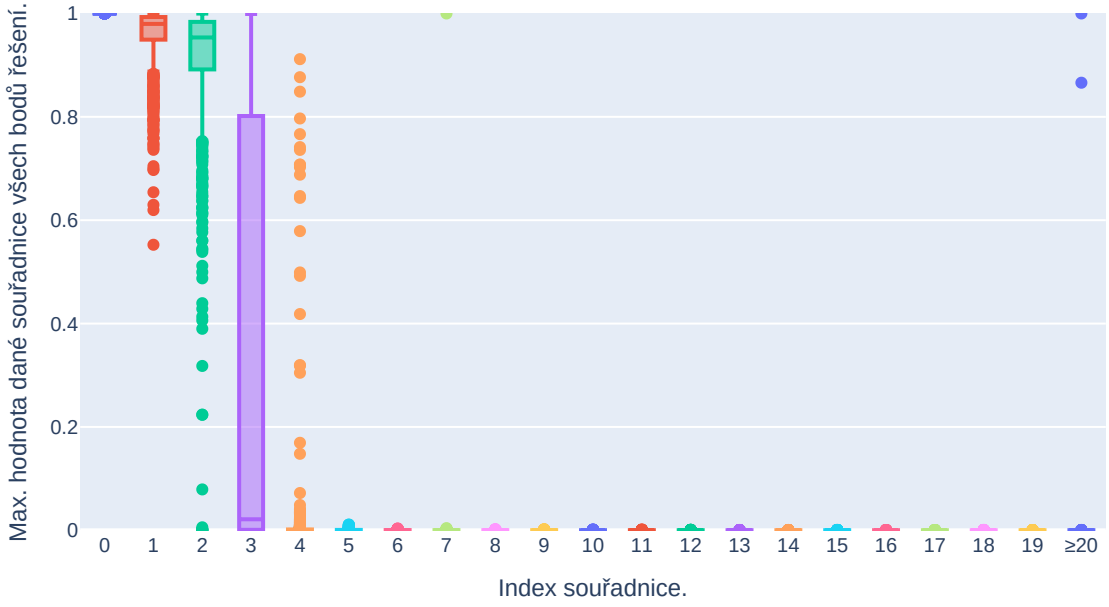
Obrázek 13: Počet vektorů s danou souřadnicí větší než 0.05 pro $n = 200$.



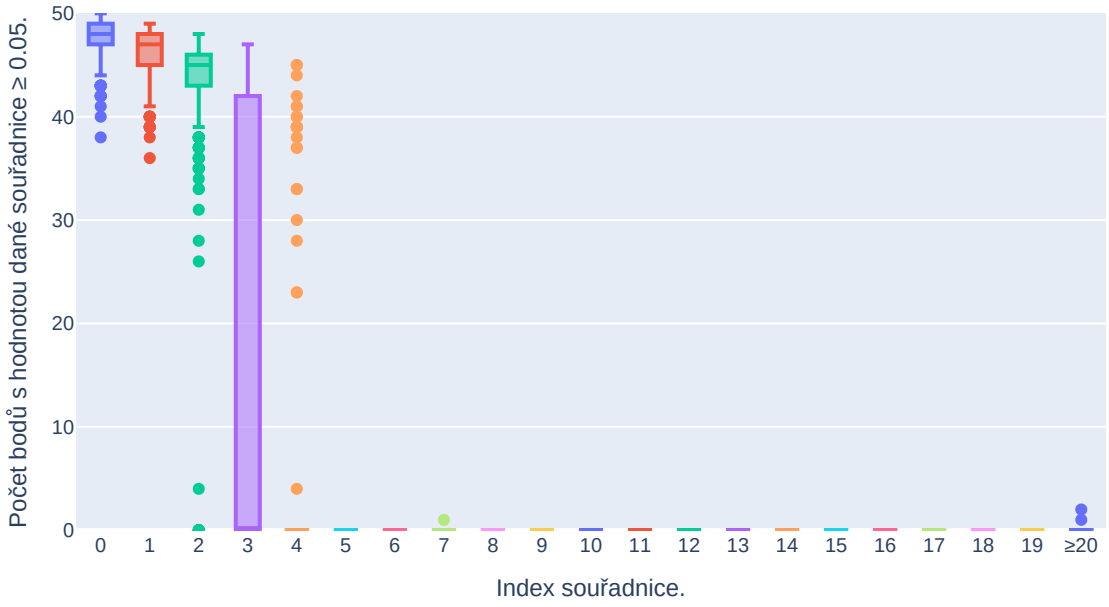
Obrázek 14: Maximální hodnota v dané dimenzi pro $n = 100$.



Obrázek 15: Počet vektorů s danou souřadnicí větší než 0.05 pro $n = 100$.



Obrázek 16: Maximální hodnota v dané dimenzi pro $n = 50$.

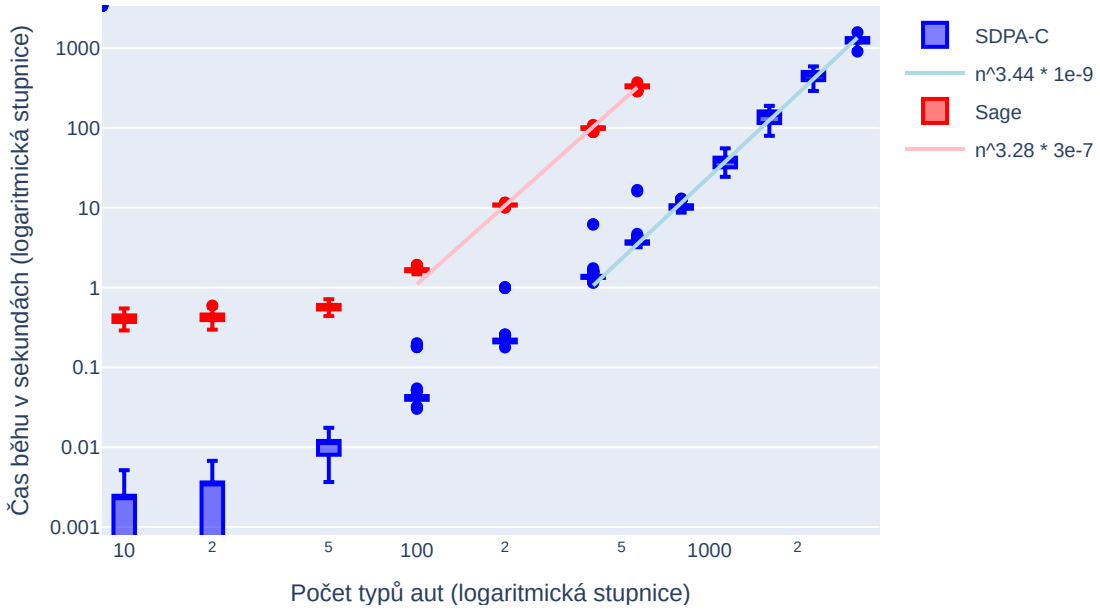


Obrázek 17: Počet vektorů s danou souřadnicí větší než 0.05 pro $n = 50$.

5.4 Časová složitost algoritmu sdp

Víme, že existuje implementace algoritmu **sdp** s polynomiální časové složitosti (za použití elipsoidové metody). Použité řešiče však využívají jiných metod řešení semidefinitních programů a ani jeden z nich moc neslibuje, jakou přesně složitost má. Proto je na místě změřit, jak rychlé řešení jsou.

Testování probíhalo na stroji s procesorem AMD Ryzen 5 7600. Programům bylo poskytnuto 16 GB operační paměti a běh byl omezen na jedno jádro procesoru. Během výpočtu na stroji neběželo nic kromě výpočtu a základních funkcí operačního systému.



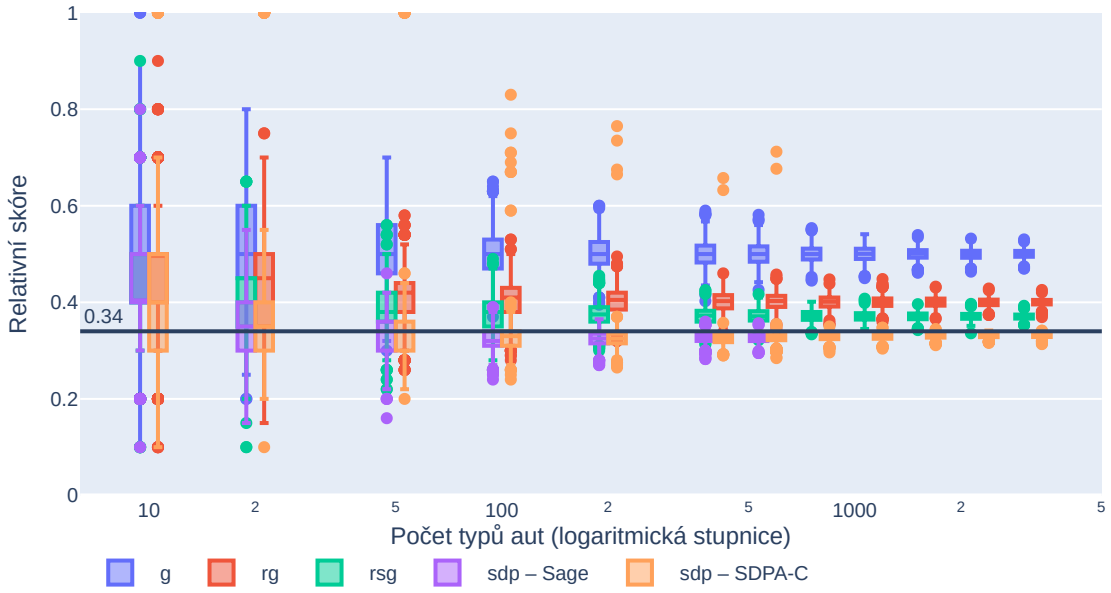
Obrázek 18: Závislost času řešení na velikosti vstupu.

Dle výše uvedeného grafu můžeme regresí v logaritmickém grafu usuzovat, že časová složitost obou algoritmů je v $\mathcal{O}(n^4) \cap \Omega(n^3)$. Implementace pomocí SDPA-C má menší multiplikativní konstantu a navíc mnohem rychlejší čas startu, který se zejména projevuje na malých instancích.

5.5 Skóre

Nejzajímavější je pro nás naměřené (relativní) skóre algoritmů.

Mezi uvažovanými řešeními jsou dvě implementace **sdp**. Implementace pomocí sage bohužel vyžaduje velké množství paměti a proto nebyla testovaná na tak velkých vstupech. O implementaci pomocí SDPA-C víme, že občas chybuje. Naštěstí chyby nenastávají tak často, aby to moc ovlivňovalo výsledné skóre. Pokud algoritmus chyboval a vrátil nějaké řešení, je uvažováno skóre daného řešení. Pokud algoritmus žádné řešení nevrátil, uvažuje se místo toho hodnota n, což je jistě horní odhad na snadno dosažitelné řešení.



Obrázek 19: Naměřená závislost relativního skóre řešení na velikosti vstupu.

Na výše uvedeném grafu a tabulkách na následujících stranách si můžeme všimnout, že s rostoucím n se u všech měřených algoritmů zmenšuje rozptyl relativního skóre.

Z grafu vidíme, že pro dostatečně velká n je naměřené relativní skóre algoritmu **sdp** menší než 0.34. Z toho můžeme tedy usuzovat, že $\delta_{\mathbf{sdp}}(n) \leq 0.34$ pro $n \in \{100, 200, 400, 566, 800, 1131, 1600, 2263\}$. Z toho pak můžeme vyslovit hypotézu, že $\delta_{\mathbf{sdp}}(n) \leq 0.34$ pro $n \geq 100$ a tedy i $\delta^{+}_{\mathbf{sdp}} \leq 0.34$.

Z naměřených dat také můžeme usuzovat, že **sdp** je lepší než libovolný z jiných představených algoritmů.

n	$\overline{\delta_{\mathbf{g}}(n)}$	$S_{\delta_{\mathbf{g}}}(n)$	5%	25%	50%	75%	95%
10	0.53	0.158	0.3	0.4	0.5	0.6	0.8
20	0.51	0.109	0.35	0.45	0.5	0.6	0.7
50	0.508	0.068	0.4	0.46	0.5	0.56	0.62
100	0.502	0.0496	0.42	0.47	0.5	0.53	0.58
200	0.501	0.0356	0.44	0.48	0.5	0.525	0.56
400	0.5	0.0258	0.46	0.482	0.5	0.517	0.542
566	0.5	0.0221	0.465	0.484	0.5	0.516	0.537
800	0.501	0.0171	0.472	0.489	0.5	0.511	0.529
1131	0.501	0.0151	0.476	0.49	0.5	0.511	0.526
1600	0.501	0.012	0.481	0.492	0.501	0.508	0.52
2263	0.5	0.0105	0.482	0.492	0.5	0.507	0.516
3200	0.5	0.00883	0.486	0.494	0.501	0.507	0.514

Veškeré hodnoty jsou zaokrouhleny na 3 platné číslice.

$\overline{\delta_{\mathbf{alg}}(n)}$ značí výběrový průměr, tedy $\frac{1}{m}\sum_{0\leq i<m} r_i$, kde m je počet testů a r_i je relativní skóre i-tého z nich.

$S_{\delta_{\mathbf{alg}}(n)}$ značí výběrovou směrodatnou odchylku, tedy $\sqrt{\frac{1}{m-1}\sum_{0\leq i<m}\left(r_i - \overline{\delta_{\mathbf{alg}}(n)}\right)^2}$.

Tabulka 1: Statistika algoritmu **g**.

n	$\overline{\delta_{\mathbf{rg}}(n)}$	$S_{\delta_{\mathbf{rg}}}(n)$	5%	25%	50%	75%	95%
10	0.46	0.131	0.3	0.4	0.5	0.5	0.7
20	0.429	0.0926	0.3	0.35	0.45	0.5	0.6
50	0.41	0.0571	0.32	0.38	0.42	0.44	0.5
100	0.406	0.0396	0.34	0.38	0.41	0.43	0.47
200	0.403	0.0284	0.355	0.385	0.405	0.42	0.45
400	0.402	0.0194	0.37	0.388	0.403	0.415	0.435
566	0.402	0.017	0.375	0.39	0.403	0.413	0.431
800	0.401	0.0139	0.378	0.391	0.401	0.41	0.422
1131	0.4	0.0121	0.38	0.393	0.401	0.408	0.42
1600	0.4	0.0103	0.384	0.393	0.401	0.407	0.417
2263	0.4	0.00835	0.387	0.394	0.4	0.406	0.413
3200	0.4	0.00714	0.388	0.396	0.4	0.405	0.412

Tabulka 2: Statistika algoritmu **rg**.

n	$\overline{\delta_{\mathbf{rsg}}(n)}$	$S_{\delta_{\mathbf{rsg}}}(n)$	5%	25%	50%	75%	95%
10	0.446	0.123	0.3	0.4	0.4	0.5	0.6
20	0.411	0.0832	0.25	0.35	0.4	0.45	0.55
50	0.387	0.0517	0.3	0.36	0.38	0.42	0.48
100	0.377	0.0353	0.32	0.35	0.38	0.4	0.43
200	0.375	0.025	0.335	0.36	0.375	0.39	0.415
400	0.372	0.0175	0.343	0.36	0.372	0.383	0.4
566	0.371	0.0142	0.346	0.362	0.371	0.382	0.394
800	0.372	0.0126	0.35	0.364	0.371	0.38	0.393
1131	0.371	0.0102	0.355	0.364	0.371	0.378	0.388
1600	0.371	0.00875	0.357	0.365	0.371	0.377	0.385
2263	0.371	0.00763	0.358	0.366	0.371	0.376	0.384
3200	0.37	0.00624	0.36	0.366	0.37	0.375	0.38

Tabulka 3: Statistika algoritmu **rsg**.

n	$\overline{\delta_{\mathbf{sdp}}(n)}$	$S_{\delta_{\mathbf{sdp}}}(n)$	5%	25%	50%	75%	95%
10	0.421	0.107	0.2	0.4	0.4	0.5	0.6
20	0.365	0.0642	0.25	0.3	0.35	0.4	0.45
50	0.33	0.0361	0.28	0.3	0.34	0.36	0.38
100	0.322	0.0242	0.28	0.31	0.32	0.34	0.36
200	0.323	0.0162	0.295	0.315	0.325	0.335	0.35
400	0.326	0.0115	0.305	0.32	0.328	0.333	0.343
566	0.326	0.00932	0.311	0.32	0.327	0.332	0.341

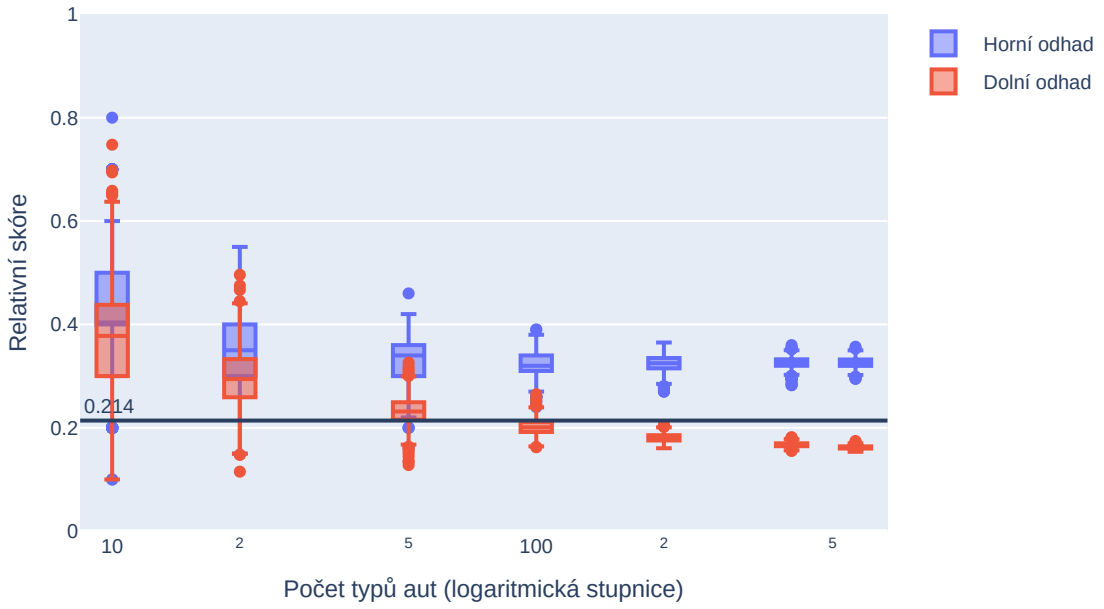
Tabulka 4: Statistika algoritmu **sdp** – Sage.

n	chyb	$\overline{\delta_{\mathbf{sdp}}(n)}$	$S_{\delta_{\mathbf{sdp}}}(n)$	5%	25%	50%	75%	95%
10	16	0.422	0.129	0.3	0.3	0.4	0.5	0.6
20	15	0.371	0.104	0.25	0.3	0.35	0.4	0.5
50	11	0.34	0.0783	0.28	0.3	0.34	0.36	0.4
100	10	0.328	0.0434	0.29	0.31	0.33	0.34	0.37
200	4	0.325	0.0291	0.295	0.315	0.325	0.335	0.345
400	2	0.325	0.0182	0.305	0.318	0.325	0.333	0.343
566	2	0.327	0.0189	0.309	0.322	0.327	0.332	0.341
800	0	0.328	0.00746	0.315	0.324	0.329	0.334	0.34
1131	0	0.329	0.00643	0.318	0.324	0.329	0.333	0.34
1600	0	0.33	0.00552	0.32	0.327	0.331	0.334	0.339
2263	0	0.331	0.00452	0.323	0.328	0.331	0.334	0.337
3200	0	0.33	0.00379	0.324	0.328	0.331	0.333	0.337

Tabulka 5: Statistika algoritmu **sdp** – SDPA-C.

5.6 Dolní odhad

Výstup z každého algoritmu nám dává horní odhad na optimum daného vstupu. Ovšem semidefinitní programování nám navíc dává i dolní odhad na optimum, protože víme, že optimum jednoho z ekvivalentních vyjádření účelové funkce je vždy menši rovno optimu BPS. Použité řešiče semidefinitních programů nám navíc dají i horní odhad na optimum SDP (který se od nalezeného řešení může nepatrně lišit).



Obrázek 20: Horní a dolní odhad na 1 000 náhodných instancí vygenerovaný sage **spd**.

Takto vygenerovaný dolní odhad nám bohužel nepřináší žádnou zajímavou informaci o chování na náhodném vstupu. Připomeňme, že Hančl a kol. [4] dokázali, že $\delta^- \geq 0.214$. Průměrný dolní odhad na testovaných vstupech je pro dostatečně velký vstup dle obrázku 20 menší než 0.214, tedy nedostáváme ani žádnou novou hypotézu na dolní odhad.

Nicméně stále je zajímavé, že pro konkrétní instanci umíme určit nějaký netriviální dolní odhad.

Závěr

V této práci jsme představili algoritmus **sdp** na BPS založený na semidefinitním programování. Bohužel se nám nepodařilo dokázat žádný netriviální odhad na $\delta^+_{\mathbf{sdp}}$. Nicméně dle naměřených dat můžeme soudit, že $\delta^+_{\mathbf{sdp}}$ se pohybuje okolo 0.34. Místo toho jsme však dokázali, že pro libovolný vstup bude střední hodnota (přes náhodná čísla generovaná algoritmem nikoliv přes vstup) skóre řešení nejhůře $0.212n$ od optima, tedy, že platí

$$\forall \alpha : \qquad \mathbb{E}[\delta_{\mathbf{sdp}}(\alpha)] \leq \delta(\alpha) + 0.212n.$$

Toto řešení jsme dále dokonce dvakrát implementovali s využitím různých implementací řešení semidefinitních programů, které jsme tímto i porovnali. Z naměřených dat jsme jednak odhadli střední hodnotu skóre algoritmu přes náhodný vstup. A také jsme si všimli, že dimenze řešení semidefinitního programování je poměrně malá, což jsme zformulovali jako hypotézu.

Stále však zůstává otevřená otázka, kolik přesně je δ^+ a δ^- (a případně zda se rovnají) a jaké nejlepší $\delta^+_{\mathbf{alg}}$ je možno dosáhnout polynomiálním algoritmem **alg**.

Seznam použité literatury

[1] Bonsma, P.; Epping, Th. a Hochstättler, W. Complexity results on restricted instances of a paint shop problem for words. *Discrete Applied Mathematics* [online]. 2006, **154**(9), 1335–1343. ISSN 0166-218X. Dostupné z: *https://www.sciencedirect.com/science/article/pii/S0166218X0500377X*

[2] Gupta, Anupam; Kale, Satyen; Nagarajan, Viswanath; Saket, Rishi a Schieber, Baruch. The Approximability of the Binary Paintshop Problem. In: Raghavendra, Prasad; Raskhodnikova, Sofya; Jansen, Klaus a Rolim, José D. P., ed. *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, s. 205–217. ISBN 978-3-642-40328-6.

[3] Andres, Stephan Dominique a Hochstättler, Winfried. Some heuristics for the binary paint shop problem and their expected number of colour changes. *Journal of Discrete Algorithms* [online]. 2011, **9**(2), 203–211. ISSN 1570-8667. Dostupné z: *https://www.sciencedirect.com/science/article/pii/S1570866710000559*

[4] Hančl, J.; Kabela, A.; Opler, M.; Sosnovec, J.; Šámal, R. a Valtr, P. Improved Bounds for the Binary Paint Shop Problem. In: Wu, Weili a Tong, Guangmo, ed. *Computing and Combinatorics*. Cham: Springer Nature Switzerland, 2024, s. 210–221. ISBN 978-3-031-49193-1.

[5] Epping, Th.; Hochstättler, W. a Oertel, P. Complexity results on a paint shop problem. *Discrete Applied Mathematics* [online]. 2004, **136**(2), 217–226. ISSN 0166-218X. Dostupné z: *https://www.sciencedirect.com/science/article/pii/S0166218X03004426*

[6] Alon, Noga. Splitting necklaces. *Advances in Mathematics* [online]. 1987, **63**(3), 247–253. ISSN 0001-8708. Dostupné z: *https://www.sciencedirect.com/science/article/pii/0001870887900557*

[7] Gärtner, Bernd a Matoušek, Jiří. *Approximation algorithms and semidefinite programming*. B.m.: Springer Science & Business Media, 2012.

[8] Sun, Kevin. *Lecture notes for Graph Algorithms, lecture 22* [online]. 2019 [vid. 2024-04-20]. Dostupné z: *https://people.csail.mit.edu/ronitt/COURSE/S20/NOTES/lec6-scribe.pdf*

[9] Dimitrakakis, Alexander. *Lecture notes for Randomness and Computation, lecture 6* [online]. 2020 [vid. 2024-04-20]. Dostupné z: *https://people.csail.mit.edu/ronitt/COURSE/S20/NOTES/lec6-scribe.pdf*

[10] SDPA Project. *SDPA Official Page* [online]. 1995–2024 [vid. 2024-03-30]. Dostupné z: *https://sdpa.sourceforge.net/*

[11] ISO/IEC. *Working Draft, Standard for Programming Language C++* [online]. 2014 [vid. 2024-04-20]. Dostupné z: *https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4296.pdf*

[12] Autoři SageMath. *SageMath Official Page* [online]. [vid. 2024-03-31]. Dostupné z: *https://www.sagemath.org/index.html*

[13] The Sage Development Team. *Sage – Semidefinite Programming* [online]. 2005-2024 [vid. 2024-03-31]. Dostupné z: *https://doc.sagemath.org/html/en/reference/numerical/sage/numerical/sdp.html*

Seznam obrázků