Ondřej Schejbal - schejond

Přiblížení zadaného úkolu

Zadáním bylo implementovat jednu z odpřednášených pokročilých iteračních technik pro řešení problému splnitelnosti booleovské formule, kde každá formule je tvořena konjunkcí klauzulí o 3 proměnných (tedy tzv. 3-SAT úloha). Měli jsme na výběr ze 3 algoritmů: simulované ochlazování, genetický algoritmus a tabu search. Rozhodl jsem se implementovat a zkoumat simulované ochlazování, s kterým mám již zkušenosti z 4. úlohy. Cílem bylo zkoumat vlastnosti vybraného algoritmu.

Problém splnitelnosti booleovské formule (SAT)

Mějme zadanou booleovskou formuli F o **n** proměnných v konjunktivní normální formě. Dále jsou dány celočíselné kladné váhy pro každou proměnnou. Cílem je najít takové ohodnocení proměnných, pro které je booleovská formule F pravdivá a součet vah proměnných, které jsou ohodnoceny jako pravdivé, byl maximální.

Kompletní zadání úlohy je na adrese: moodle-vyuka.cvut.cz/mod/assign/view.php?id=89703

Popis implementace

Cílem mé implementační fáze bylo vytvořit funkční návrh, který bude prakticky využitelný pro libovolný algoritmus a bude snadné ho rozšířit pro jakoukoliv verzi SAT problému. Tento cíl se mi povedlo splnit a vytvořil jsem třídy tak, že vhodně zachycují a vyhodnocují stav zadané booleovské formule. Jejich přiblížení je v následující sekci. Model je možné jednoduše rozšířit na jiné instance SAT problému, jako je například MAX SAT (optimalizační varianta úlohy s cílem nalézt co nejvíce splněných klauzulí). Implementace také není omezená na zadaný 3-SAT problém a umožňuje pracovat s libovolným počtem proměnných v klauzulích, počty proměnných mohou dokonce být v každé klauzuli jiné. Váhy, které jsou proměnným přiděleny jsou uloženy tak, aby umožňovali okamžité přiřazení pro požadovanou proměnnou a zároveň neomezovali možnost realizace problému, který s vahami nepočítá.

Model

V této části jsou přiblíženy detaily tříd, které se v mé implementaci vyskytují.

Formula - třída uchovává všechny základní dostupné informace o problému. Obsahuje id dané instance problému, počet proměnných, počet klauzulí, list vah pro jednotlivé proměnné a také obsahuje list klauzulí, které se v ní vyskytují. Nad objektem této třídy je možné zavolat metodu *evaluate*, která z přijatého vektoru, obsahující ohodnocení pro jednotlivé proměnné, je schopna vyhodnotit počet splněných klauzulí . Pokud pro tuto formuli je známo optimální řešení, tak je umožněno toto řešení uložit v rámci objektu.

Clause - třída pro reprezentaci jedné klauzule. Obsahuje mapu, kde klíčem je id proměnné a hodnota je informace o tom, zda se v dané klauzuli vyskytuje daná proměnná v normální podobě, nebo je přítomna její negace. I samotná klauzule se umí pomocí své metody *evaluate* vyhodnotit pro zadaný vektor ohodnocení proměnných a je tedy možné vyhodnotit pouze vybrané klauzule. To je vhodné a používám to v situacích, kdy například měním ohodnocení pouze jedné proměnné a vím, že před touto změnou byla celá formula vyhodnocena jako pravdivá. Potřebuji tedy znovu vyhodnotit pouze ty klauzule, ve kterých se vyskytuje proměnná, jejíž hodnota se změnila.

Solution - třída, jejíž objekty reprezentují jednotlivé instance možných řešení. Obsahuje Id relevantní instance problému, list boolean hodnot, jež reprezentuje podobu řešení. Každý prvek z listu reprezentuje ohodnocení proměnné, jež odpovídá indexu prvku. Třída také obsahuje proměnné, které informují o tom, zda je pro danou podobu řešení formule pravdivá, či není. Také obsahuje proměnnou s celkovou vahou tohoto řešení a proměnnou indikující počet splněných klauzulí pro dané ohodnocení.

Algorithm - abstraktní třída, která definuje základní rozhraní pro algoritmy a jejich pomocné metody, jako je například práce s měřením času. Toto rozhraní má za cíl usnadnit přidání dalších algoritmů. Přidání dalšího algoritmu tedy vyžaduje pouze samotnou implementaci a potřebná režie je obstarána zděděním z této abstraktní třídy.

DataLoader a DataSaver jsou třídy, která slouží pro manipulaci se vstupními a výstupními soubory. Na vstupu počítá má implementace s úlohou SATu ve formátuMWCNF, který nám byl představen pro tuto úlohu a jsou v této podobě datové sady, které nám byly zprostředkovány.

Přiblížení implementovaných algoritmů

BruteForce

BruteForce je obecný algoritmus, který reprezentuje hledání řešení problému za pomocí hrubé síly. Jinými slovy vyzkouší všechny kombinace a díky tomu najde řešení. Pro náš problém to znamená vyzkoušet celkem $2^{\mathbf{n}}$ možných ohodnocení jednotlivých proměnných, kde **n** je počet proměnných pro zadanou instanci problému. Ve své implementaci využívám bitového posunu k vyzkoušení všech možných kombinací. Algoritmus jsem implementoval z toho důvodu, že plánuji experimentovat s instancemi, které si sám vygeneruji a pro možnost naměření hodnoty relativní chyby zkoumaného algoritmu potřebuji znát i skutečnou správnou hodnotu optimálního řešení.

Simulované ochlazování

Simulované ochlazování je algoritmus založený na jednoduché heuristice, která je obohacena o techniku, která ze simulovaného ochlazování dělá poměrně výkonný algoritmus. Hlavním cílem tohoto algoritmu je snaha vylepšit heuristické hledání řešení tím, že se pokouší předejít uváznutí v lokálním optimu během hledání nejoptimálnějšího řešení. Tato snaha je založena na jednoduchém principu.

Algoritmus prohledává stavový prostor optimalizačního problému. Začíná ve vytvořeném počátečním stavu a v každém kroku vybere náhodného souseda a porovná kvalitu jeho řešení s kvalitou řešení pro aktuální pozici. Pokud soused přináší zlepšení, přejde do něj, pokud nepřináší, tak za určité pravděpodobnosti se do jeho stavu algoritmus také rozhodne přejít. Právě tato pravděpodobnost a možnost přijmutí přechodu do horšího stavu umožňuje algoritmu vyřešit problém uváznutí v lokálním optimu. Vzorec pro tuto pravděpodobnost je založen na aktuální hodnotě parametru teploty, více o něm dále.

V algoritmu se nachází několik parametrů a možností, které ovlivňují jeho běh. Zde jsou vlastnosti těchto parametrů pouze přiblíženy společně s obecnou strukturou kódu algoritmu. Postup jejich získávání a konkrétní volby je popsán v dalších částech.
Algoritmus vychází z počátečního stavu, neboli instance kandidáta na řešení. Má již předem nastavené hodnoty parametrů pro počáteční teplotu, koncovou teplotu a parametr pro řízení ochlazování. Hlavní tělo algoritmu tvoří dvojitý cyklus. Během popisu budu hovořit o dvou proměnných, které si uchovávám. Tzv. globální nejlepší řešení a tzv. lokální nejlepší řešení. Na počátku algoritmu jsou obě dvě tyto proměnné rovny počátečnímu stavu.

Vnitřní cyklus běží vždy pro aktuální hodnotu teploty a opakuje prohledávání sousedů. Vždy vygeneruje náhodný index a pro lokální nejlepší řešení převrátí hodnotu proměnné odpovídající danému indexu. Následně porovná vlastnosti takto vygenerovaného souseda s lokálním nejlepším řešením.

Porovnání vlastností jednotlivých řešení

Pro SAT je potřeba vhodně zvolit porovnávací kritérium jednotlivých instancí řešení. Při konstrukci algoritmu a následném testování jsem narazil na to, že volba tohoto kritéria má na celkový výkon algoritmu velký vliv. Níže jsou popsány má pozorování.

- Nejprve jsem vyzkoušel porovnávat pouze součet vah kladně ohodnocených proměnných u jednotlivých řešení. To se ukázalo jako špatná volba, protože algoritmus často ani nenašel žádné správné řešení a když ano, tak to bylo převážně díky vhodně vygenerovanému počátečnímu stavu, neboli správnost řešení závisela čistě na náhodě.
- Následně jsem začal počítat počet splněných klauzulí v zadané formuli pro konkrétní ohodnocení proměnných. To vedlo k drobnému zlepšení, ale výsledky byly stále velmi špatné (velikost relativní chyby byla kolem 40 %) a situace s častým nenalezením žádného validního řešení se opakovala.
- Nakonec se mi osvědčilo tyto dva přístupy výše zkombinovat. Nejprve porovnám počet splněných klauzulí. Pokud se počty splněných klauzulí nerovnají, tak pro porovnávané stavy **A** a **B** definuji **δ=(počet splněných klauzulí pro A) - (počet splněných klauzulí pro B)**. Pokud stavy mají počet klauzulí schodný, tak definuji **δ**jako **δ=(suma vah kladně ohodnocených proměnných v A) - (suma vah kladně ohodnocených proměnných v B)**. Na základě hodnoty **δ** se následně jednoduše rozhodne, který stav je lepší (pro příklad se stavy A a B by kladná hodnota δ znamenala, že stav A je lepší). Hodnota δ je následně použita při rozhodování, zda přijmout horší řešení, více o tom popisuji dále.

Pokud je po porovnání stavů nový stav vyhodnocen jako lepší, pak ho algoritmus přijme a aktualizuje na něj hodnotu lokálního nejlepšího řešení. Pokud má tato nalezená instance lepší vlastnosti i než globální nejlepší řešení, aktualizuje se i to. Pokud je nová instance vyhodnocena jako horší než lokální nejlepší řešení, tak se vygeneruje náhodné desetinné číslo (double) z intervalu [0, 1) a to se porovná s hodnotou **δ** (získání její hodnoty je popsáno o odstavec výše) následovně:

```
random.nextFloat() < Math.exp(δ/t)
```

, kde t je rovna aktuální teplotě. Hlavním trikem celého algoritmu je, aby hodnota na pravé straně nerovnice byla schopna na základě teploty přijímat horší řešení s rozumnou pravděpodobností, která spolu se snižující se teplotou bude klesat. Tohoto stavu se mi povedlo dosáhnout díky vhodnému výpočtu hodnoty δ. Tento vnitřní cyklus běží buďto 2*__k__-krát, kde **k** je počet klauzulí v dané formuli a nebo běží do té doby, než proběhne celkem **k** posunů po stavovém prostoru.

Vnější cyklus má na starosti spouštění cyklu vnitřního, vhodnou inicializaci parametrů počítajících iterace a má také na starost aktualizaci hodnoty teploty. Teplota má při prvním běhu cyklu hodnotu rovnou parametru počáteční teploty. Vždy po skončení vnitřního cyklu se aktualizuje hodnota aktuální teploty pomocí vzorce $t = t*\alpha$, kde t je aktuální teplota a α je hodnota parametru, který řídí ochlazování. Vnitřní cyklus běží do té dobu, než teplota dosáhne, nebo se sníží pod úroveň hodnoty parametru konečné teploty, nebo do té doby, než se nejlepší globální řešení aktualizuje **k***100-krát, kde **k** je počet klauzulí v dané formuli. Druhá podmínka má za úkol zastavit algoritmus v případě, že je nejlepší hodnota aktualizována neúměrně často, tedy nejpravděpodobněji nastala ve stavovém prostoru situace, kdy přebíháme mezi dvěma srovnatelnými sousedy.

Postup při hledání vhodných hodnot parametrů pro Simulované ochlazování

Algoritmus je ve svém principu obecný a lze jej nasadit na různé problémy. Pro každý problém je ovšem vhodná jiná kombinace parametrů. Pro problém splnitelnosti booleovské formule jsou některé parametry jasné. Například omezení počtu cyklů v závislosti na počtu klauzulí je zcela přirozené. Větší množství iterací by totiž často implikovalo opakované porovnávání některých stavů. K tomuto závěru jsem dospěl po vyzkoušení několika konstant.

Pro algoritmus je ale nejvíce stěžejní přijít na nejvhodnější hodnotu těchto tří parametrů: počáteční resp. koncová teplota (označme **t_0** resp. **t_final**) a parametr řídící rychlost chlazení (značíme **α**). Rozhodl jsem se tedy provést tzv. pilotní experiment nad třemi instancemi a podrobně pozorovat chování algoritmu a jeho reakce na změnu parametrů. Cílem bylo nalézt nejvhodnější hodnoty těchto parametrů a následně pozorovat jejich chování pro různé datové sady. Je obecná pravda, že pro každou instanci lze najít jiné vhodné hodnoty parametrů, ale mým cílem bylo tyto parametry dobře odladit pro vybrané instance a následně pozorovat, jak si algoritmus poradí i s jinými, složitějšími instancemi za použití těchto parametrů.

Pro pozorování skutečné citlivosti algoritmu na změnu parametrů jsem danou instanci nechal algoritmem vyhodnotit 200 krát a pozoroval průměr naměřených hodnot relativních chyb výpočtů. Díky tomu mohu argumentovat, že výsledky, které algoritmus vrací nejsou závislé na náhodě, ale algoritmus k nim vždy dospěje. Níže popisuji mnou vybrané a otestované hodnoty a význam jednotlivých parametrů.

Parametr α

Parametr α byl jeden z nejlehčích na experimentování. Jeho doporučená a nejvíce používaná hodnota je z intervalu (0.8, 1). Algoritmus určuje rychlost ochlazování a při hodnotách blížících se 0.8 teplota až moc rychle klesala a algoritmus kvůli tomu velmi často skončil v lokálním optimu. Pro hodnoty blížící se 1 sice algoritmus provádí více iterací, ale také mnohem více horších instancí dostalo díky vyšším teplotám šanci a díky tomu byla šance na uváznutí mnohem menší. Hodnoty blížící se 1 také ukazovali nejlepší výsledky. Výsledná hodnota α, s kterou jsem dosahoval nejlepších výsledků a použil ji je 0.96.

Parametry t_0 a t_final

Parametry t_0 a t_final mi zabraly nejdelší čas vyladit. Tyto parametry mají na kvalitu výsledků nejvyšší vliv. Hlavním cílem bylo zvolit hodnotu počáteční teploty takovou, aby v kombinaci s hodnotou **δ** byla procentuální šance na přijetí horšího řešení v rozumné hodnotě. Mým cílem, který se mi povedlo uskutečnit, bylo, aby počáteční teplota vytvářela v průměru 50% hranici na přijetí horšího řešení (horší řešení s velmi dobrými vlastnostmi měla samozřejmě hodnotu na přijetí vyšší) a tato hranice s ubývající teplotou rovnoměrně klesala.

Vzhledem k tomu, že nikde není definováno, jakých hodnot má počáteční teplota nabývat jsem experimentoval s hodnotami z intervalu [100, 10 000]. Vypozoroval jsem, že nízké počáteční teploty nejsou vhodné, protože algoritmus provede malé množství iterací a malá hodnota teploty má nepříznivý vliv na procentuální hranici přijetí horšího výsledku. Konkrétně způsobuje, že skoro každé horší řešení je přijato. Vysoká hodnota počáteční teploty sice zařídila velký počet iterací, ale výpočty algoritmu byly zbytečně dlouhé kvůli pomalému ochlazování. Nakonec jsem jako nejvhodnější interval pro volbu počáteční teploty vytyčil interval [1 100, 1 600]. Z čehož při mých experimentech nejlépe vycházela hodnota průměrné relativní chyby pro hodnotu t_0 = 1 500.

U parametru t_final , určující konečnou teplotu, velké hodnoty způsobovaly, že mi algoritmus doběhl dříve než stačil nalézt alespoň trochu dobré řešení. Pro extrémně malé hodnoty zase způsoboval zbytečné navýšení počtu iterací, které zdržovalo algoritmus opět bez přínosu lepších výsledků. Přitom při nízkých teplotách již šance na přijetí horšího řešení byla tak malá, že skoro žádné horší řešení nebylo nikdy přijato. Nakonec jsem ukotvil hodnotu tohoto parametru na hodnotě t_final = 0.1.

Výše testované hodnoty jsem odladil na náhodně vybraných instancích problému splnitelnosti booleovské formule s 20 proměnnými a 78 klauzulemi (každá klauzule obsahuje 3 proměnné, viz zadání), které všechny pocházely z datové sady M1, kterou jsme dostali pro tento úkol připravenou a se známými výsledky. Konkrétně se jednalo o instance *01000*, *0101* a *0103*. Jelikož se stále jedná o aproximační algoritmus, který prohledá jen zlomek stavového prostoru, je dost běžné, že nemusí najít optimální řešení, ovšem relativní chyba by měla být malá. Finální hodnoty parametrů jsem tedy zvolil tak, aby tuto relativní chybu měli co nejmenší. Instanci jsem nechal algoritmus vyhodnotit 200 krát a zprůměroval naměřenou relativní chybu u jednotlivých běhů, abych získal pravý vliv parametrů. Pro mnou vybrané parametry jsem na zkoumaných instancích dosahoval v průměru hodnot relativní chyby ~0.07 %. Jeden z pozorovaných běhů je zachycen na obrázku pod tímto odstavcem, kde je možné vidět, že algoritmus z 200 běhů nedospěl ke správným výsledkům pouze ve 4 případech, což je velmi dobrý výkon.

```
START
Relative error in found solution best price: 0,01614
Relative error in found solution best price: 0,01812
Relative error in found solution best price: 0,01614
Relative error in found solution best price: 0,01812
Average best price mismatch in 200 executions of pilot test: 0,00034
END
```

Pro dobrou představu a ujištění se o kvalitě zvolených parametrů jsem nechal vyhodnotit všechny instance z datové sady M1 pro 20 proměnných a 78 klauzulí. Průměrná relativní chyba byla pro zvolené parametry rovna 0.181 %, což je dle mého názoru velmi dobrý výsledek. Drobné zhoršení je, vzhledem k tomu, že jsem při ladění parametrů s těmito instancemi nepočítal, očekávané a hodnota relativní chyby je stále natolik malá, že mohu i tak výsledek považovat za velmi dobrý.

Počáteční stav

Kromě parametrů jsem experimentoval i s různými volbami výchozího stavu. Celkem jsem zkoušel 3 možnosti

- všechny proměnné nastavit jako false
- všechny proměnné nastavit jako true
- nastavení počátečních hodnot proměnných náhodně pro každý běh

Výsledky všech těchto 3 přístupu vycházely v průměru srovnatelně. Vzhledem k tomu, že varianta "všechny true" a varianta "všechny false" jsou dle mého názoru lepší pouze pro konkrétní instance, tak jsem se rozhodl ve své implementaci použít tu nejobecnější variantu, která na podobě instance není závislá ani trochu. Ve své implementaci mám tedy variantu, kde nastavuji počáteční hodnoty proměnných náhodně pro každý běh.

Vylepšení algoritmu a vyzkoušené pokročilé techniky

Kromě základní podoby algoritmu je v mé implementaci možné vidět, že jsem přišel na velmi efektivní způsob porovnávání kvality jednotlivých stavů a výpočtu hodnoty δ. Rozhodl jsem se, že také vyzkouším známé pokročilé techniky pro algoritmus simulovaného žíhání (tzv. adaptační techniky). Níže vždy stručně přiblížím princip každé z nich a podobu po nasazení do mé implementace.

První z aplikovaných adaptačních technik byl tzv. **teplotní restart**. Pokud po vychladnutí teploty nebyla nalezená konfigurace řešením, pro jehož ohodnocení proměnných by byla formule splnitelná, tak se teplota restartuje a algoritmus zkouší najít vhodné řešení znovu. To vyžadovalo přidání dalšího while cyklu, který vše obaloval a hlídal, konkrétně jsem využil výhod do-while cyklu. Jelikož případy, ve kterých algoritmus nenalezl validní řešení byly velmi výjimečné, tak tato technika nepřinesla přímé zlepšení. Je pravda, že ale pomohla vyřešit tyto výjimečné případy, protože často, při druhém pokusu o nalezení řešení pro danou instanci byl algoritmus již úspěšný. Maximální počet teplotních restartů jsem nastavil na 3.

Druhou technikou, kterou jsem vyzkoušel bylo navyšování počtu iterací vnitřního cyklu postupně se snižující se teplotou. Po každém ochlazení teploty jsem hodnotu omezující počet vnitřních cyklů (originálně na hodnotě 2***k**, kde **k** je počet klauzulí) vynásobil konstantou 1.001. Tuto hodnotu jsem zvolil takto nízkou kvůli tomu, že aktualizací teploty je mnoho a hlavní myšlenkou není drasticky zvyšovat počet iterací. To by velmi zvýšilo časovou náročnost algoritmu. Vybraná hodnota nezvyšovala časovou náročnost algoritmu nijak razantně a zároveň byla schopna při nižších teplotách přidat větší počet iterací. Bohužel tato technika nepřinesla zlepšení a ukázala se tedy být pro mou implementaci neefektivní, komplexitu mé implementace zbytečně zvyšovala. To byl i důvod proč jsem ji do finální podoby mého kódu nezahrnul.

Měření a výsledky algoritmu na zadaných sadách

Pro pozorování chování algoritmu jsem nejprve sledoval naměřené hodnoty při zpracování instancí z datových sad, které nám byly pro tuto úlohu poskytnuty. Jednalo se o tyto sady:

- M1, N1: instance, které by měli být jednoduché na výpočet
- R1, Q1: instance, které by měli být složitější na výpočet, protože zadané váhy poměrně komplikují nalezení optimálního řešení
- A1: velmi obtížné úlohy na výpočet

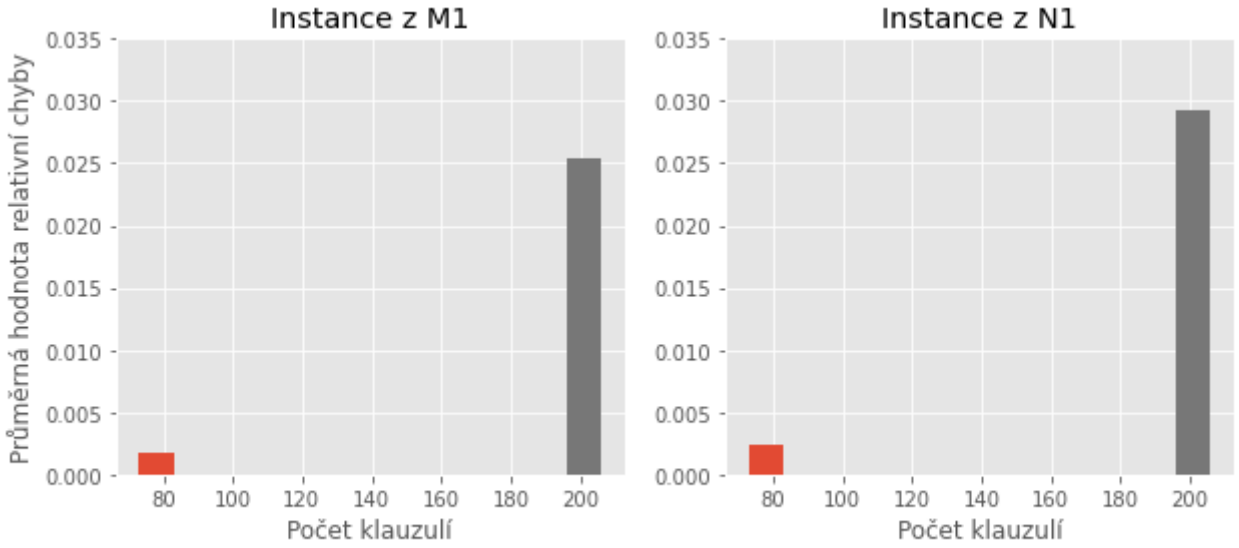
Sady M1, N1, R1 a Q1 obsahují vždy dva druhy instancí. Instance s 20 proměnnými a 78 klauzulemi a instance s 50 proměnnými a 201 klauzulemi. Jednotlivé druhy instancí se liší ve svých počtech, proto jsem sledoval hlavně průměrné naměřené hodnoty. Sada A1 obsahovala také dva druhy instancí a to instance s 20 proměnnými a 88 klauzulemi a instance s 20 proměnnými a 91 klauzulemi.

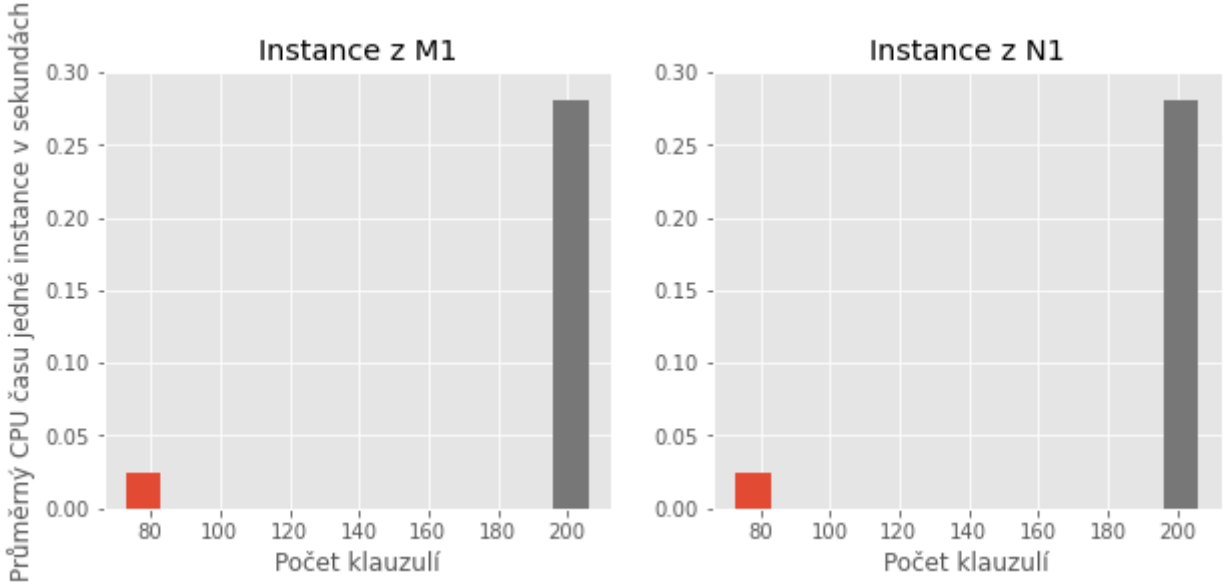
Experimenty jsem realizoval v jazyku Java na notebooku s 64bitovým operačním systémem Windows 10 a procesorem Intel i-5-6300U @ 2.40GHz.

Čas výpočtů, který jsem měřil a používal, je tzv. čas strávený na vlákně procesoru. Viz:oktech.hu/kb/profiling-java-application-measuring-real-cpu-time

V grafech níže je možné vidět naměřené hodnoty průměrného CPU času v sekundách a průměrné naměřené relativní chyby. Porovnány jsou vždy jednotlivé dvojice, jak jsou představeny výše (sada A1 je tedy zobrazena sama o sobě). Grafy jsou vždy následované tabulkou zachycující naměřené maximální a minimální hodnoty relativní chyby.

Sady M1 a N1

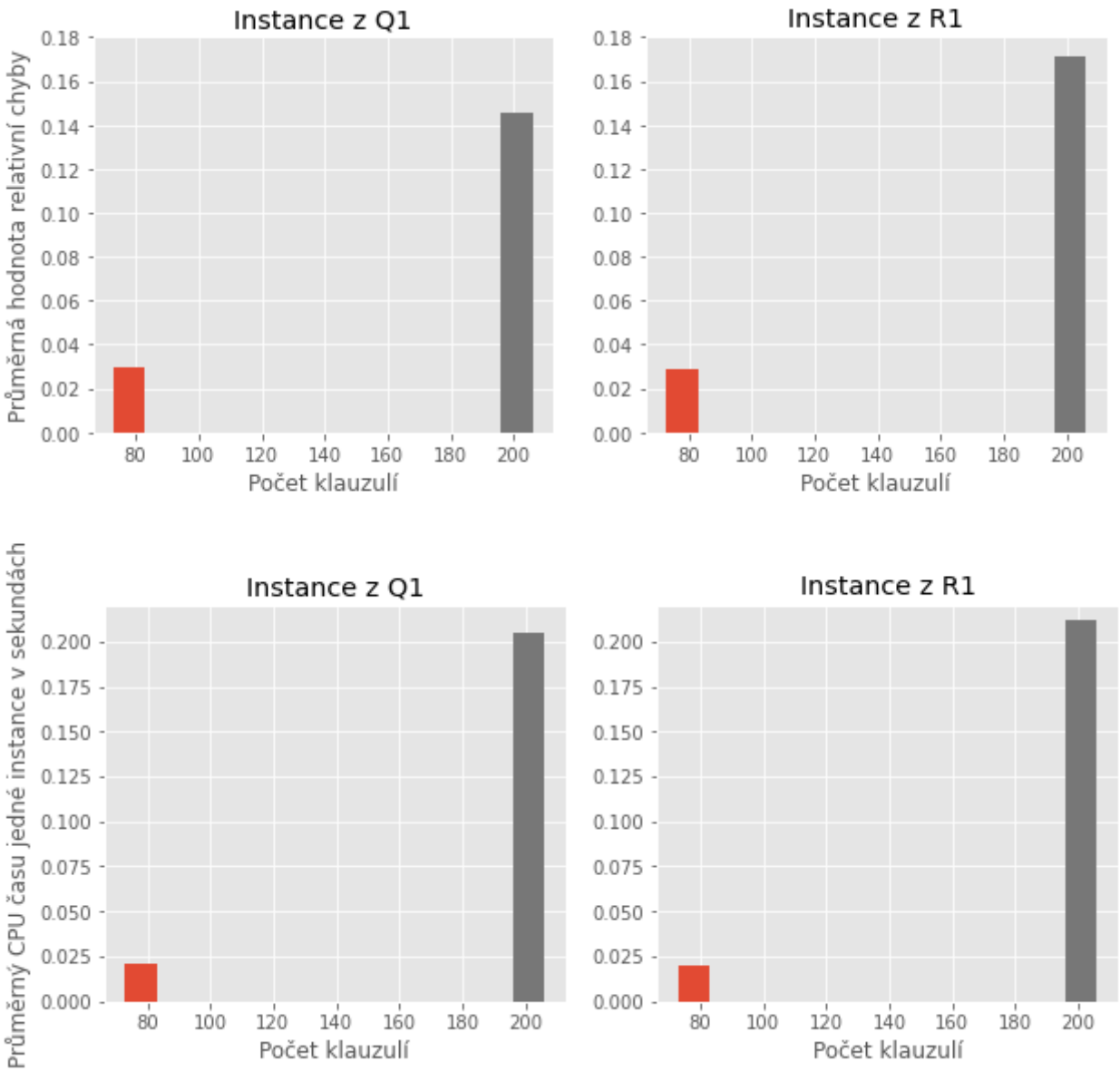




	Min. relativní chyba	Max. relativní chyba

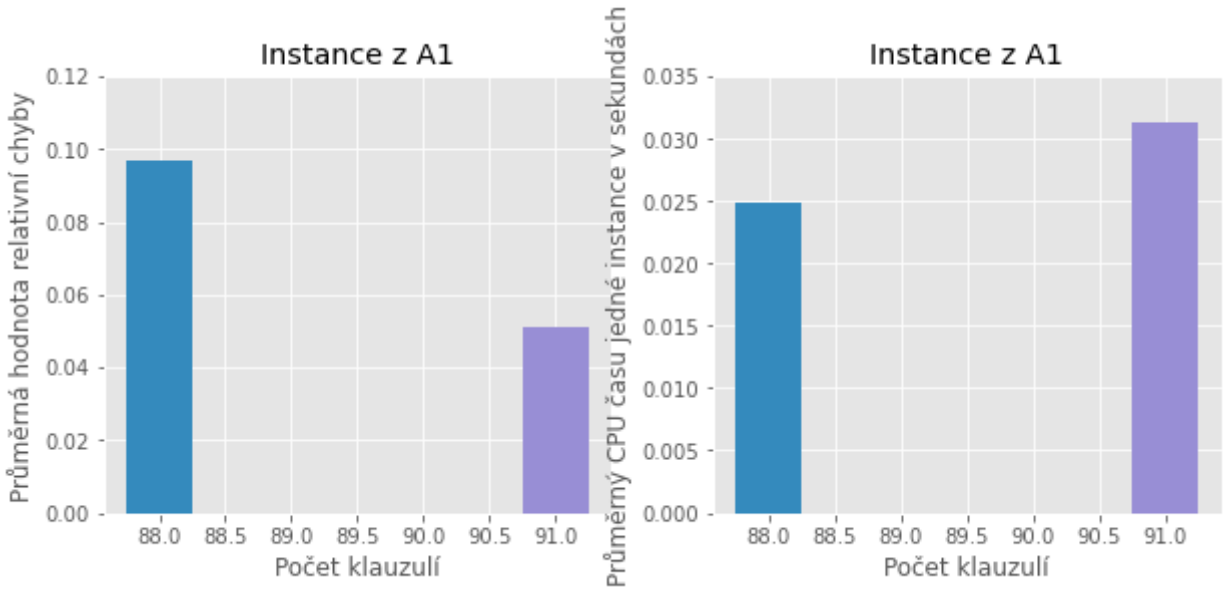
M1 - 20 proměnných - 78 klauzulí	0	0.137356
M1 - 50 proměnných - 201 klauzulí	0	0.223069
N1 - 20 proměnných - 78 klauzulí	0	0.205964
N1 - 50 proměnných - 201 klauzulí	0	0.291686

Sady Q1 a R1



	Min. relativní chyba	Max. relativní chyba
Q1 - 20 proměnných - 78 klauzulí	0	0.334970
Q1 - 50 proměnných - 201 klauzulí	0	0.752789
R1 - 20 proměnných - 78 klauzulí	0	0.413942
R1 - 50 proměnných - 201 klauzulí	0	0.754985

Sada A1



	Min. relativní chyba	Max. relativní chyba
A1 - 20 proměnných - 88 klauzulí	0	0.636363
A1 - 20 proměnných - 91 klauzulí	0	0.805970

V grafech je možné pozorovat například, že relativní chyba stoupá úměrně s náročností instance. Na instancích M1, kde byly parametry odlaďovány dosahuje velmi nízké chyby a u nejtěžší datové sady A1 má hodnota o řád horší výsledky. V grafech je možné vidět, že relativní chyba stoupá také s počtem zadaných proměnných. U druhého druhu instancí ze sady A1 (20 proměnných + 91 klauzulí) je zajímavé, že hodnota průměrné relativní chyby je nižší než při stejném počtu proměnných, ale méně klauzulích. Z toho usuzuji, že relativní chyba je závislá na počtu proměnných a ne na počtu klauzulí.

Velmi dobrý výsledek zaznamenal algoritmus ve statistice minimální relativní chyby. Je možné vidět, že i pro složitější instance je schopen v některých případech dokonvergovat ke korektnímu řešení. Z maximálních naměřených hodnot je zase možné vidět, že rostou společně s počtem zadaných proměnných.

Vlastní experimenty citlivosti algoritmu

Rozhodl jsem se provést i experimenty na jiných než zadaných sadách, abych mohl sledovat citlivost algoritmu na různé vlastnosti instancí. K tomu jsem použil generátor 3-SAT instancí, který vytvořil M. Motoki a je volně dostupný například z tohoto repositáře na GitHubu: github.com/6d2f892d053abd018ef3. Generátor umí vygenerovat instance 3-SAT problému a má možnost nastavení následujících 3 parametrů:

- počet proměnných - budu značit **n**
- počet klauzulí - budu značit **k**
- maximální váhy proměnné - budu značit **w**

Jelikož generátor nevrací optimální řešení pro vygenerované instance, je potřeba toto řešení nejdřív najít. Kvůli tomu jsem implementoval algoritmus BruteForce. Pomocí něj vždy najdu korektní řešení (i když za cenu větší časové náročnosti) a díky tomu jsem pak schopen měřit hodnotu relativní chyby algoritmu simulovaného ochlazování.

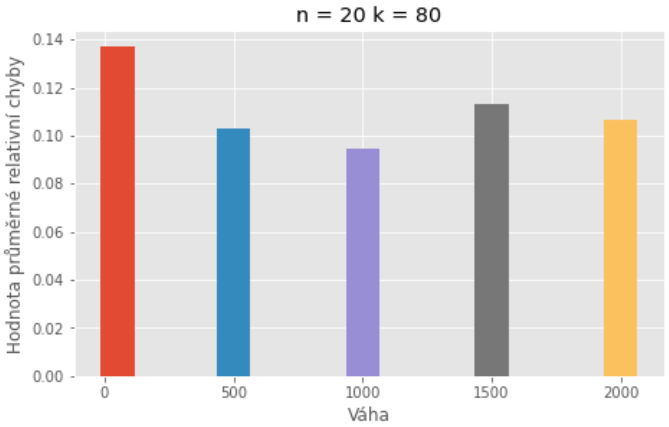
Na základě předchozích naměřených hodnot jsem si vytvořil 3 hypotézy, kde každá z nich se týká citlivosti algoritmu na jeden z parametrů generátoru. Níže v jednotlivých sekcích tyto hypotézy přiblížím a představím výsledky většího experimentu. Na základě naměřených hodnot tohoto velkého experimentu následně hypotézu buďto potvrdím, nebo vyvrátím.

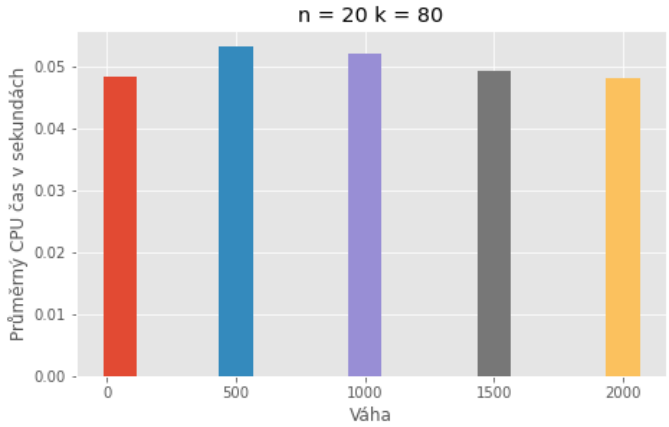
Každá kombinace parametrů, která je zmíněna a zachycena na grafech níže je založena na hodnotách naměřených pro 200 vygenerovaných instancích.

Vliv parametru omezující maximální váhu proměnné

Parametr váhy nastavuje maximální hodnotu, které může nabývat jedna proměnná. Dle mých pozorováních a hlavně z obecného principu algoritmu se domnívám, že výkon a časová složitost algoritmus nebude různými hodnotami tohoto parametru ovlivněna.

Výsledky níže jsou pro náhodně vygenerované instance, které měly všechny pevně dané a stejné hodnoty parametru **n** a **k** a to 20, resp. 80. Instance v experimentu nabývají hodnoty parametru **w** 50, 500, 1 000, 1 500 a 2 000.

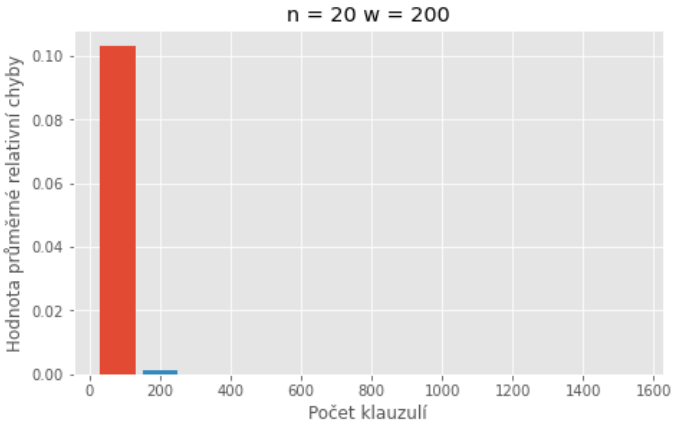


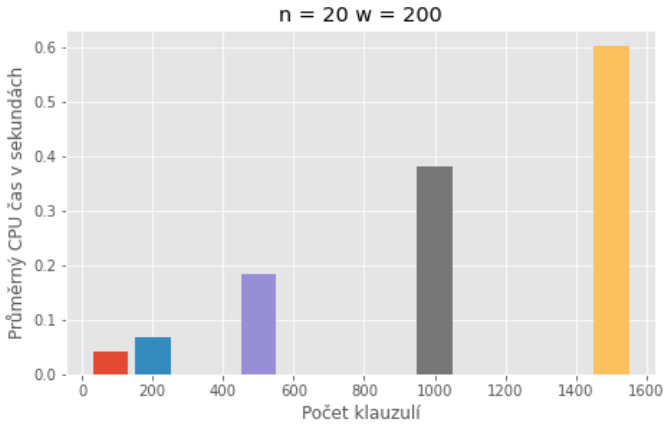


Z naměřených hodnot je možné vidět, že se má hypotéza potvrdila. Hodnota relativní chyby i doba výpočtu jsou s drobnými výchylkami schodné pro různé hodnoty parametru **w**. Maximální hodnota váhy tedy nemá na výkon algoritmu přímý vliv.

Vlivu počtu klauzulí

Na základě výsledků na datových sadách, které jsme dostali k dispozici jsem vypozoroval, že se zvyšujícím se počtem klauzulí stoupá časová složitost, ale relativní chyba mírně klesá se zvyšujícím se počtem klauzulí (viz. hodnoty naměřené na instancích ze sady A1). Tuto hypotézu ověřuji na instancích s **n** = 20 a **w** = 200. Parametr **k** nabývá hodnot 80, 200, 500, 1 000 a 1 500.

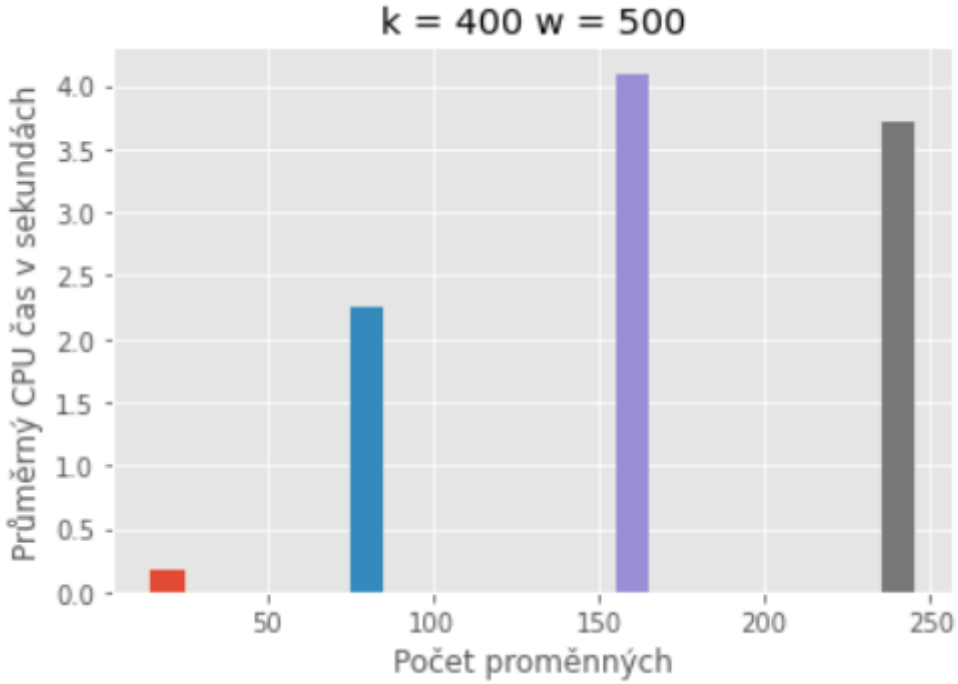




Na výsledcích je možné vidět, že se má hypotéza potvrdila. K mému překvapení se hodnota relativní chyby snižovala velmi prudce společně s velkých počtem klauzulí. Čekal jsem pouze pozvolné klesání, ale v prvním grafu je možné vidět, že pro velké hodnoty parametru **k** je již naměřená hodnota průměrné relativní chyby nesrovnatelná s hodnotami naměřenými pro malou hodnotou parametru **k**. Hypotéza týkající se zvyšující časové složitosti se mi také potvrdila.

Vliv počtu proměnných

Pro parametr **n** určující počet proměnných jsem z minulých měření vypozoroval, že se zvyšujícím se počtem proměnných stoupá výpočetní časová složitost a zároveň stoupá i hodnota relativní chyby. Vzhledem k tomu, že pro každou vygenerovanou instanci potřebuji nejprve nalézt korektní řešení pomocí algoritmu BruteForce, nemohl jsem otestovat velké rozpětí hodnot (velké hodnoty **n** trvaly algoritmu dlouhé hodiny, než se povedlo nalézt řešení). Proto jsem v testech pozoroval pouze průměrnou časovou náročnost výpočtu pro různá **n**. Hypotézu týkající se časové složitosti se pokusím ověřit na experimentu s instancemi, kde parametry **k** a **w** jsou rovny 400, resp. 500 a parametr **n** nabývá hodnot 20, 80, 160 a 240.



Jedna část hypotézy se mi tedy nepovedla potvrdit kvůli omezení ze strany výpočetního výkonu. Ve výsledcích zobrazených výše je možné vidět, že část hypotézy týkající se časové složitosti se mi potvrdit povedla a je možné ji v grafu pozorovat.

Zjištění výkonu algoritmu

Na závěr jsem se rozhodl ověřit, pro jak velké instance je schopna má implementace algoritmu najít řešení v rozumné době. Konkrétně jsem stanovil limit 15 minut pro výpočet. Zjistil jsem, že algoritmus je schopen zvládnout instance až o 1 000 proměnných a 5 000 klauzulích v tomto limitu.

Závěr

Úspěšně se mi povedlo sestavit funkční model, které je připravena na práci s verzemi a instancemi problému SAT a je ji možné kdykoliv lehce rozšířit. Úspěšně jsem implementoval algoritmus BruteForce a algoritmus simulovaného ochlazování. U simulovaného ochlazování jsem vyzkoušel pár adaptačních technik, z nichž jsem jednu považoval za užitečnou a ponechal ji i ve finální verzi mé implementace. Dále jsem úspěšně nalezl nejvhodnější hodnoty parametrů pro algoritmus simulovaného ochlazování a ty jsem následně podrobil důkladných testům na sadách, které nám byly k této úloze zprostředkovány. Na výsledcích jsem si ověřil, že díky mým nalezeným parametrům má algoritmus poměrně dobré výsledky i pro složité instance. Také jsem úspěšně otestoval citlivost algoritmu simulovaného ochlazování na počtu klauzulí, proměnných a na maximální váze proměnné pomocí instancí, které jsem si sám vygeneroval. Některé hypotézy o citlivosti algoritmu jsem na základě těchto experimentů potvrdil, jiné vyvrátil.

Ondřej Schejbal

gitlab.fit.cvut.cz/schejond/ni-kop/tree/master/KOP-5