Paralelní SAT řešič

Úloha

Tento program je složen z několika částí. Hlavní částí je řešič SAT problémů, který odpovídá na otázky „Je tato booleovská formule v konjunktivní normální formě splnitelná, případně jak?“. Řešič lze volat s použitím více vláken či jen jako sekvenční algoritmus.

Další části jsou některé další NP-úplné problémy, které lze na problém SAT převést. Tyto problémy jsou tedy řešeny převedením na SAT a použitím řešiče. Výsledek řešiče je poté interpretován a je vrácena vhodná odpověď. Konkrétně jsou implementovány následující problémy:

- **problém nezávislé množiny v grafu**, kdy se uživatel ptá, zda v grafu existuje nezávislá množina vrholů alespoň tak velká, jako zadané číslo *k*.
- **problém obarvení grafu třemi barvami**, kdy se uživatel ptá, zda lze vrcholy grafu obarvit třemi barvami tak, aby žádný z dvou vrcholů, které sdílejí hranu, neměly stejnou barvu.
- **problém hamiltonovské cesty**, kdy se uživatel ptá, zda v grafu existuje taková cesta, která obsahuje každý vrchol grafu právě jednou.

Algoritmy

SAT řešič

U SAT řešiče jsem zvolil algoritmus **DPLL**, který je založen na backtrackingu. Tento algoritmus obecně funguje tak, že se vybere nějaký literál (tomuto místu výběru se říká „místo větvení“), přiřadí se mu nějaká hodnota, čímž se zjednoduší daná formule, a rekurzivně se kontroluje, zdali je formule stále splnitelná. Pokud je, tak je i původní formule splnitelná. Pokud ovšem není, pak se algoritmus vrací zpět na výběr hodnoty literálu a přiřadí mu opačnou hodnotu, než předtím.

Tento algoritmus se dá ovšem vylepšit použitím dalších 2 pravidel v každém kroku. První pravidlo je „**unit propagation**“. Toto pravidlo říká, že pokud je klauzule jednotková, tedy obsahuje jediný literál, pak lze tomuto literálu přiřadit hodnotu tak, aby byla jednotková klauzule splněna.

Dalším pravidlem je „**pure literal elimination**“. Toto pravidlo říká, že pokud je literál „**pure**“ (český ekvivalent by mohl být „ryzí“), pak mu lze přiřadit hodnotu tak, aby byl ve všech klauzulích splněn (tedy s hodnotou „pravda“). „Pure“ literál je takový, který se ve formuli vyskytuje pouze s jedním znaménkem (negace nebo positivně). Tím se tedy zbavíme všech klauzulí, které tento literál obsahují.

Původně jsem tento algoritmus implementoval rekurzivně, jak je popsáno výše, ale v některých případech nastávalo **přetečení zásobníku**. Algoritmus jsem tedy pouze převedl na iterační přidáním datové struktury *Stack<>*, do které jsem přidával formule / (parciální) modely, které čekaly na vyřešení. Pokud byl model na vrcholu nesplnitelný, vyhodil se. Pokud se zásobník vyprázdnil, je nesplnitelná i původní zadaná formule. Na zásobník se formule přidávají v místě větvení.

Existuje i jiný algoritmus zvaný **CDCL**, který je určitým vylepšením algoritmu DPLL. Je však složitější na implementaci a mnohem složitější na paralelizaci než DPLL.

Paralelizace

Co se týče paralelizování tohoto algoritmu, bylo potřeba se snažit nějakým způsobem **práci rozdělit rovnoměrně mezi všechna vlákna**. Kdybychom pouze vytvářeli nová vlákna v místě větvení a jedno vlákno by pokračovalo s jedním ohodnocením literálu a druhé s tím opačným, často by se stávalo, že jedno vlákno by dostalo za úkol řešit mnohem větší podstrom než to druhé, které by mohlo třeba v dalším kroku zjistit, že je formule již nesplnitelná. Navíc vytváření nových vláken a neužívání znovu těch samých značně zpomaluje program.

Proto se na začátku vytvoří příslušný počet pracovních vláken (tolik, kolik je k dispozici logických procesorů), které sdílejí frontu, do které postupně vždy v místě větvení přidávají další modely (podstromy) k řešení. Hlavní vlákno pak čeká dokud ho jedno z pracujících vláken neprobudí, že už má výsledek, nebo dokud se nezjistí, že fronta je prázdná a žádné další vlákno už nepracuje. V obou případech jedno „náhodné“ pracovní vlákno ukončí ty ostatní a informuje hlavní vlákno o připraveném výsledku a samo skončí. Jde vlastně o variantu modelu producent-konzument, kde každé pracující vlákno je zároveň producentem i konzumentem. Vše je řízeno zámky a konstrukcemi *Monitor.Wait* a *Monitor.Pulse*.

Nezávislá množina

Algoritmus řešení nezávislé množiny v grafu je, jak jsem popsal výše, následující: převést graf na CNF formuli, která bude splnitelná právě tehdy, když v zadaném grafu bude existovat nezávislá množina velikosti alespoň *k*, a ověřit splnitelnost této formule pomocí řešiče.

Převod grafu na CNF formuli probíhá tak, že pro každý vrchol budeme mít literál v_n, který bude indikovat to, jestli byl daný vrchol vybrán do nezávislé množiny. Chceme nezávislou množinu, tedy přidáme klauzule $(!v_i \mid\mid !v_j)$ pro každou hranu *(i,j)* v grafu.

Dále potřebujeme, aby byla nezávislá množina alespoň tak velká, jako *k*. Představíme si tedy matici literálů tvaru $k \times n$, kde *n* je počet vrcholů v grafu. X_{ij} znamená, že *i*-tý prvek nezávislé množiny je vrchol *j*.

Musíme tedy přidat klauzule $(!X_{ij} \mid\mid !X_{kj})$ pro *i != k*, které zařídí, aby v každém sloupci matice byla nejvýše jedna „pravda“.

Dále musíme přidat klauzule $(!X_{ij} \mid\mid !X_{ik})$ pro *j != k* a klauzule $(X_{i0} \mid\mid X_{i1} \mid\mid \ldots \mid\mid X_{in})$, což zařídí aby v každém řádku matice byla právě jedna „pravda“.

Nakonec musíme přidat klauzule $(!X_{ij} \mid\mid v_j)$, které zajistí vztah mezi očíslováním a nezávislou množinou.

3-obarvenost

Algoritmus řešení tohoto problému je vlastně stejný jako v předchozím případě, jen se liší převod.

Nechť jsou vrcholy grafu očíslovány od 1 do n. Pro každý vrchol *i* grafu přidáme klauzule *(i-Red || i-Green || i-Blue)*, což znamená, že každý vrchol má jednu ze tří barev.

Dále přidáme *(!i-Red || !i-Green)*, *(!i-Red || !i-Blue)*, *(!i-Green || !i-Blue)*, což zajistí to, že každý vrchol bude mít přiřazenu jednu barvu.

Zbývá přidat klauzule, které zajistí, aby dva sousední vrcholy neměly stejnou barvu. Přidáme pro každou hranu *(i,j)* v grafu klauzule *(!i-Red || !j-Red)*, *(!i-Green || !j-Green)*, *(!i-Blue || !j-Blue)*.

Hamiltonovská cesta

Zde převod na SAT problém probíhá takto:

Vytvoříme proměnné X_{ij} , které znamenají, že na *i*-té pozici v hamiltonovské cestě je vrchol *j*.

Zavedeme klauzule $(X_{1j} \mid\mid X_{2j} \mid\mid \ldots \mid\mid X_{nj})$ pro každé *j*, což zajistí, že každý vrchol *j* se někde objeví na hamiltonovské cestě.

Dále se nesmí žádný vrchol *j* na cestě objevit více než jednou: $(!X_{ij} \mid\mid !X_{kj})$ pro *i != k*.

Dále každá pozice *i* na cestě musí být obsazena nějakým vrcholem: $(X_{i1} \mid\mid X_{i2} \mid\mid \ldots \mid\mid X_{in})$.

Žádné dva vrcholy *j* a *k* nesmí být na stejné pozici v cestě: $(!X_{ij} \mid\mid !X_{ik})$ pro *j != k*.

Nakonec vrcholy *i* a *j*, které nemají společnou hranu, nemohou být v cestě vedle sebe: $(!X_{ki} \mid\mid !X_{k+1j})$.

Program

CNF

Klíčovou datovou strukturou je třída ***CNF***, která reprezentuje samotnou formuli v CNF. Po vyřešení reprezentuje model dané formule. Při řešení zase parciální model. Obsahuje ***List<Clause> Clauses***, což je seznam reprezentující klauzule ve formuli. Dále také obsahuje ***List<Variable> Variables***, což je zase seznam všechn unikátních proměnných ve formuli.

Objekt typu *CNF* lze instanciovat pomocí **3 konstruktorů**. Jeden bezparametrový, jeden s parametrem *List<Clause>* a poslední s parametrem *CNF*, který udělá hlubokou kopii objektu vloženém v parametru.

Dále lze na objektu typu CNF volat například **metody**:

void ReadFormula(TextReader reader), která přečte z parametru vstup (více o vstupech později),

string ToString(), která vrací string, který shrnuje hodnoty všech proměnných, a

string GetInputFormat(), která vrací string, který reprezentuje možnou variantu původního vstupu.

Poznámka: Pokud objekt typu *CNF* vložíme jako parametr řešiči, pak nemá smysl tento samý objekt znovu vkládat jako parametr řešiči, jelikož tento objekt už má model (přiřazené hodnoty). To se hodí vědět například při benchmarkingu, pokud chci porovnat paralelní a sekvenční řešič – musím nejprve před řešení vytvořit dva objekty *CNF*, do jednoho načíst data a do druhého data zkopírovat konstruktorem přijímajícím *CNF*.

Clause

Třída ***Clause*** reprezentuje jednu klauzuli ve formuli v CNF. Obsahuje ***List<Variable>*** **Variables**, což je seznam všech proměnných v dané klauzuli. Objekt tohoto typu lze instanciovat **konstruktorem s jedním parametrem** *List<Variable>*.

Na objektu typu *Clause* lze volat například **metody**:

bool IsTrue(), která vrátí *true*, pokud je v klauzuli alespoň jedna proměnná s přiřazenou hodnotou a její výsledná hodnota je *true*.

bool IsFalse(), která vrátí *true*, pokud mají v klauzuli všechny proměnné přiřazenou hodnotu a jejich výsledná hodnota je *false*.

Variable

Třída ***Variable*** reprezentuje jednu proměnnou ve formuli v CNF. Obsahuje **datové položky**, které proměnnou charakterizují:

bool Sign – znaménko proměnné (negace nebo positivní)

bool Value – přiřazená hodnota

bool IsAssigned – Je proměnné přiřazena nějaká hodnota?

string Name – identifikátor proměnné

Objekt typu *Variable* lze instanciovat pomocí **2 konstruktorů**, jeden přijímá parametry ***bool Sign, string Name*** a druhý všechny zmíněné datové položky.

Na objektu tohoto typu lze volat například **metody**:

bool GetFinalValue(), která vrátí výslednou hodnotu proměnné (bere v potaz znaménko a hodnotu).

void SetValue(CNF cnf, bool Value), která nastaví hodnotu všem proměnným stejného jména ve formuli *cnf* na hodnotu *Value*.

DPLL

Další klíčovou datovou strukturou je třída ***DPLL***, která reprezentuje instanci DPLL algoritmu. Má dvě privátní **datové položky**:

Queue<CNF> sharedModelQueue – sdílená fronta pro CNF (parciální modely) k vyřešení pro kterékoliv vlákno

bool parallel – indikuje, zda má algoritmus využívat multithreading

Lze instanciovat **konstruktorem bez parametrů**.

Zde jsou klíčové **instanční metody**:

DPLLResultHolder SatisfiableParallel(CNF cnf) – vyřeší cnf paralelně a vrátí výsledek (více níže) (interně každé vlákno volá metodu o řádek níže)

DPLLResultHolder Satisfiable(CNF cnf) – vyřeší sekvenčně cnf a vrátí výsledek

void Branch(Stack<CNF> stack, Variable variable) – tato je privátní a je volaná uvnitř metody *Satisfiable*;

reprezentuje místo větvení, tedy nastaví *variable* na *false* v původním modelu, vytvoří nový model, ve kterém zase nastaví *variable* na *true*;

dále dle datové položky *parallel* buď přidá nově vytvořený model do *stacku* nebo do *sharedModelQueue*

Variable GetPureVariable(CNF cnf) – privátní metoda volaná v *Satisfiable,* vrátí „pure“ proměnnou z formule *cnf*

Variable GetUnitClause(CNF cnf) – privátní metoda volaná uvnitř *Satisfiable*, vrátí jednotkovou klauzuli

Variable GetUnassigned(CNF cnf) – privátní metoda volaná uvnitř *Satisfiable*, vrátí první nepřiřazenou proměnnou

DPLLResultHolder

Tato třída obaluje výsledek algoritmu DPLL. Obahuje datové položky:

bool SAT – je formule splnitelná?

CNF model – pokud je splnitelná, tak je zde výsledný model

Na této třídě lze pak volat metodu ***string ToString()***, která vypíše výsledek. Výsledek je buď to, že je formule nesplnitelná, a nebo že je splnitelná s následující formulí (přesněji o výstupech a vstupech později).

SolverThread

Tato třída obaluje třídu *Thread*. Obsahuje navíc všechny potřebné reference a hlavní funkci vlákna, ve které probíhá téměř veškerá synchronizace a vznikají signály o výsledku.

Třídy reprezentující další NP-úplné problémy

Dále jsou v programu třídy ***IndependentSetProblem, ThreeColorabilityProblem*** a ***HamiltonianPathProblem***. Všechny mají společné to, že mají datovou položku typu ***Graph*** a funkce:

void ReadInput(TextReader reader) – přečte vstup (který reprezentuje graf)

CNF ConvertToCNF() – provede výše zmíněné algoritmy převodu daného problému na SAT a vrátí CNF formuli

string InterpretDPLLResult(DPLLResultHolder result) – vrátí string, který popisuje výsledek řešiče a tím shrne výsledek řešeného problému

Graph

Jednoduchá třída popisující graf. Lze na něm volat metodu ***void ReadGraph(TextReader reader)*** a obsahuje ***List<Vertex> Vertices***.

Vertex

Jednoduchá třída popisující vrchol. Obsahuje ***string Id*** a seznam sousedů.

Jak kód používat – demo

SAT solver

Pokud chceme řešit nějakou CNF formuli, je nejprve nutné instanciovat objekt typu *CNF* a přečíst vstup.

```
CNF cnf = new CNF();
cnf.ReadFormula(Console.In)     // místo Console.In může být jakýkoliv TextReader
```

Dále je potřeba instanciovat objekt typu *DPLL*.

```
DPLL dpll = new DPLL();
```

Nakonec můžeme zavolat funkci *Satisfiable(Parallel)* a uložit výsledek:

```
DPLLResultHolder result = dpll.SatisfiableParallel(cnf)
```

Výsledek je možné vypsat třeba na konzoli:

Console.WriteLine(result);

Ostatní problémy

Pokud chceme řešit nějaký z dalších problémů, je potřeba instanciovat příslušnou třídu. Napíši příklad pro problém nezávislé množiny (ostatní jsou stejně až na jméno třídy problému).

Nejprve tedy vytvoříme instanci třídy *IndependentSetProblem* a zavoláme funkci, která přečte vstup:

IndependentSetProblem problem = new IndependentSetProblem();
problem.ReadInput(Console.In);

Dále zavoláme funkci na převod do CNF a uložíme výsledek:

CNF problemCNF = problem.ConvertToCNF();

Poté vyřešíme stejně jako u SAT řešiče. Pokud ale budeme chtít vypsat srozumitelný výsledek, je potřeba to udělat následovně. Nechť *problemDPLLResult* je objekt vrácený funkcí *Satisfiable* ve třídě *DPLL*. Pak výsledek vypíšeme následovně:

Console.WriteLine(problem.InterpretDPLLResult(problemDPLLResult));

Jak program používat aneb Vstupy a výstupy

Při spuštění je po uživateli vyžadována volba, jestli program ukončit, či volba problému.

„Exit program (0), SAT solver (1), Independent set problem (2), 3-Colorability problem (3), Hamiltonian path problem (4)

Choose an option (number): _“

Uživatel by měl zadat číslo v závorce podle toho, co chce zvolit (0 a jakékoliv jiné nenabídnuté číslo ukončí program) a zmáčknout *Enter*.

Volba SAT solver (1)

Po zvolení čísla 1 – SAT solveru je uživateli vypsán návod, jak formátovat vstup. Je potřeba zadat booleovskou formuli v DIMACS formátu, tedy takto:

Veškeré řádky začínající „c “ značí **komentáře**. Tyto řádky jsou ignorovány.
První řádek, který není komentářem musí mít následující formát: „*p cnf **v** **c***“, kde ***v*** je přirozené číslo označující počet unikátních proměnných a ***c*** je přirozené číslo označující počet klauzulí.

Následují samotné **klauzule**. Ty se skládají **z celých čísel** oddělenými mezerami a **každá klauzule musí být ukončena nulou**. Pro přehlednost se doporučuje mít každou klauzuli na jednom řádku, ovšem lze vše mít například na jednom řádku. Celá čísla reprezentují jména proměnných. **Žádná z proměnných se nemůže jmenovat „0“** z výše zmíněného důvodu. Záporné číslo reprezentuje negaci proměnné. Poznámka: Proměnné mohou být i řetězce, pokud to nebude řetězec „c“ a již zmíněná „0“.

Po zadání příslušného počtu klauzulí se začne formule řešit. Po vyřešení je uživateli vypsán vstup v následujícím formátu:

Pokud je formule splnitelná, je vypsán řetězec „*Satisfiable with model*:“ a na každé další řádce řetězec typu „***v: bool***“, kde ***v*** je jméno proměnné a ***bool*** je buď *true* nebo *false* podle hodnoty proměnné.
V opačném případě je vypsán řetězec „*Unsatisfiable*.“.

Příklad vstupu a výstupu pro formuli (not 1 or 2 or 3) and (2 or not 3):

Vstup:
p cnf 3 2
-1 2 3 0
2 -3 0

Výstup:
Satisfiable with model:
1: False
2: True
3: False

Volba Independent set problem (2)

Po zvolení této možnosti je uživateli opět vypsán návod, jak formátovat vstup:

První řádek musí být **přirozené číslo**, které reprezentuje, jak velkou nezávislou množinu alespoň chceme.

Další vstup je pro neorientovaný **graf** v následujícím formátu:
- První řádek je posloupnost kladných čísel oddělených mezerami **začínající číslem 1** a pokračující vždy číslem o jedna větší. Tato posloupnost označuje všechny vrcholy grafu.
- Další řádky jsou vstupem pro hrany. Pro každou hranu jeden nový řádek s dvěmi kladnými čísly, která již byla zadána v předchozí posloupnosti, a jsou oddělena mezerou. Takovýto řádek označuje hranu mezi prvním číslem (vrcholem) na řádku a druhým (není potřeba zadávat hranu opačným směrem).
- Vstup se poté ukončí prázdným řádkem.

Následuje vyhodnocení a vypsání výsledku. Pokud taková nezávislá množina v grafu neexistuje, pak je vypsán řetězec „*Such independent set does not exist*.“.
V opačném případě je vypsán řetězec „*The following vertices can be chosen for the independent set of size atleast k:“*, kde *k* je první zadané číslo.
Následují řádky, každý s jedním názvem vrcholu (jedním číslem), což jsou vrcholy, které lze vybrat do nezávislé množiny.

Příklad vstupu a výstupu:

Vstup: *2*
1 2 3
1 2
1 3

Výstup: *The following vertices can be chosen for the independent set of size atleast 2:*
2
3

Volba 3-colorability problem (3)

Při této volbě je po uživateli žádán vstup grafu (viz popis Independent set problem volby).

Po vstupu se začne vyhodnocovat. Následuje výstup tvaru „***v** is **Colour***“, kde ***v*** je číslo vrcholu a ***Colour*** je jedna ze tří barev „Red“, „Green“, „Blue“, pokud je graf 3-obarvitelný. V opačném případě program vypíše řetězec „Not 3-colorable.“.

Příklad vstupu a výstupu:

Vstup: *1 2 3*
1 2
1 3

Výstup: *1 is Red*
2 is Blue
3 is Blue

Volba Hamiltonian path (4)

Při této volbě je opět žádán vstup grafu (viz popis výše).

Po zadání vstupu se vstup vyhodnotí a vypíše se výsledek tvaru „***p.** position is vertex **v***“, kde ***p*** je kladné číslo popisující pozici v cestě a ***v*** je číslo vrcholu, v případě, že v grafu existuje hamiltonovská cesta. V opačném případě vypíše řetězec „No hamiltonian path exist.“.

Příklad vstupu a výstupu:

Vstup: *1 2 3*
1 2
1 3

Výstup: *1. position is vertex 3*
2. position is vertex 1
3. position is vertex 2

Benchmark SAT řešiče

BenchmarkDotNet=v0.11.5, OS=Windows 10.0.17134.885 (1803/April2018Update/Redstone4)
Intel Pentium CPU N3710 1.60GHz, 1 CPU, 4 logical and 4 physical cores
Frequency=1562452 Hz, Resolution=640.0197 ns, Timer=TSC
[Host] : .NET Framework 4.7.2 (CLR 4.0.30319.42000), 32bit LegacyJIT-v4.7.3416.0
DefaultJob : .NET Framework 4.7.2 (CLR 4.0.30319.42000), 32bit LegacyJIT-v4.7.3416.0

Method	file	Mean	Error	StdDev
SequentialSAT	155Vars_1135Clauses_SAT.txt	2.548 s	0.0106 s	0.0099 s
ParallelSAT	155Vars_1135Clauses_SAT.txt	1.070 s	0.0213 s	0.0421 s
SequentialSAT	42Vars_133Clauses_UNSAT.txt	1.868 s	0.0037 s	0.0034 s
ParallelSAT	42Vars_133Clauses_UNSAT.txt	1.012 s	0.0116 s	0.0109 s
SequentialSAT	350Vars_1349Clauses_SAT.txt	1.147 s	0.0031 s	0.0027 s
ParallelSAT	350Vars_1349Clauses_SAT.txt	7.243 s	0.2181 s	0.6430 s
SequentialSAT	350Vars_1349Clauses_UNSAT.txt	7.011 s	0.0227 s	0.0213 s
ParallelSAT	350Vars_1349Clauses_UNSAT.txt	2.486 s	0.0216 s	0.0202 s

Z výsledků je patrné, že ve většině testovaných případů byl rychlejší paralelní řešič, což se dalo očekávat. Při testování na malých formulích je ovšem rychlejší sekvenční, jelikož overhead vytváření vláken a zamykání zámků je příliš velký.