SAT-solver

Programmeeropdracht 6
Algoritmen en Complexiteit 2018

Deze opdracht is uitgespreid over twee weken. De opdracht telt 2x zo zwaar mee in vergelijking met de andere programmeeropdrachten.

1 Probleembeschrijving

Het vervulbaarheidsprobleem (SAT) gaat over het waar maken van logische formules met conjuncties, disjuncties en negaties. Gegeven een logische formule, is het mogelijk om een waarheidswaarde voor alle variabelen te vinden, zodanig dat de formule waargemaakt wordt.

Neem als voorbeeld de formule $(\neg C \vee A \vee \neg B) \wedge (\neg B \vee C \vee \neg A)$. Een voorbeeld van een valuatie die deze formule waar maakt is A = 1 B = 0 en C = 1. Er zijn in dit geval ook andere toewijzingen van de variabelen mogelijk die de formule waar maken. Daarnaast kan het ook voorkomen dat een formule helemaal niet vervulbaar is, bijvoorbeeld $\neg A \wedge \neg A$.

In deze programmeeropdracht ga je een programma schrijven dat gegeven een formule een toekenning van de variabelen geeft zodanig dat de formule waar wordt (als dat mogelijk is).

SAT is echter een NP-volledig probleem, dus een volledig efficiënt algoritme is niet mogelijk (hiermee zou je aantonen dat P=NP). Je mag dus beginnen met een brute-force algoritme om dit probleem op te lossen. Het is echter nog steeds wel mogelijk om optimalisaties toe te passen om het probleem sneller op te lossen. In deze opdracht ga je ook deze optimalisaties implementeren om je solver sneller te maken.

De eerste stap om het probleem überhaupt op te kunnen lossen is door een verifier te schrijven. Dit programma moet simpelweg gegeven een logische formule en een toewijzing van waarden aan variabelen (bijv. $A = 1, B = 0$...) kunnen antwoorden of deze toewijzing inderdaad de formule waar maakt.

Vervolgens kun je een programma schrijven dat daadwerkelijk een dergelijke toewijzing gaat proberen te zoeken. Om te controleren of je antwoord klopt, kun je gebruikmaken van de verifier die je al geschreven hebt.

2 Invoer/uitvoer van de verifier

2.1 Invoer

We nemen aan dat de logische formule altijd in CNF is. Dat wil zeggen, de logische formule is een grote conjunctie van disjuncties en negaties komen alleen voor op variabelen. Op de eerste regel van de invoer staat één getal: het eerste getal geeft aan hoeveel variabelen k er in de logische formule staan en het tweede getal geeft aan hoeveel conjuncties c er in de logische formule aanwezig zijn. Vervolgens volgen er k regels waarin de variabelen staan samen met de valuatie die ze krijgen. Tot slot volgen c regels waarin de logische formule volgt. Elke regel is een apart deel van de conjunctie en de variabelen op die regel zijn de conjunctie. Een geeft de negatie aan. De formule in het voorbeeld hieronder komt dus overeen met $(A \vee \neg B) \wedge (C \vee D \vee \neg A) \wedge (B \vee \neg C)$.Het is ook mogelijk dat in plaats van bovenstaande er alleen de tekst "Niet vervulbaar" wordt meegegeven (en dus niks anders). In dat geval moet je als output ook “Niet vervulbaar” geven. De reden dat je dit geval apart moet toevoegen is dat de output van de solver als input van de verifier gebruikt wordt.

2.2 Uitvoer

Als uitvoer wordt de tekst “Vervult formule” verwacht als deze toewijzing de formule waarmaakt, anders wordt "Vervult formule niet" als output verwacht.

2.3 Voorbeeldinvoer

```
4 3
A 1
B 1
C 0
D 1
A ~B
C D ~A
B ~C
```

2.4 Voorbeelduitvoer

```
Vervult formule
```

3 Invoer/uitvoer van de solver

3.1 Invoer

De invoer van de solver is vergelijkbaar met de invoer van de verifier. Het verschil tussen de twee is dat de verifier wel een valuatie meekrijgt, terwijl een solver deze zelf moet zien te bedenken. Op de eerste regel van de invoer staan twee getallen: het eerste getal geeft aan hoeveel variabelen k er in de logische formule staan en het tweede getal geeft aan hoeveel conjuncties c er in de logische formule aanwezig zijn. Daarna volgen er k regels waarin de namen van de variabelen staan. Vervolgens volgen c regels waarin de logische formule volgt. Elke regel is een apart deel van de conjunctie en de variabelen op die regel zijn de conjunctie. Het tildeteken geeft de negatie aan. De formule in het voorbeeld hieronder komt dus overeen me $(A \vee \neg B) \wedge (C \vee D \vee \neg A) \wedge (B \vee \neg C)$.

3.2 Uitvoer

De uitvoer wordt op zo een manier gemaakt dat de uitvoer van de solver rechtstreeks als invoer voor de verifier gebruikt kan worden. Als uitvoer wordt op de eerste regel het aantal variabelen k en het aantal conjuncties c verwacht. Vervolgens volgen k regels met de toewijzing van de variabelen zodanig dat deze de hele formule waarmaakt (NB: je hoeft er maar één en deze kan ook anders zijn dan de voorbeelduitvoer). Tot slot volgen c regels met daarin de logische formule (exact zoals deze ook in de input staat). In het geval de formule op geen enkele manier waargemaakt kan worden, geef je als output in plaats van de bovengenoemde regels alleen de tekst “Niet vervulbaar”.

3.3 Voorbeeldinvoer

```
4 3
A
B
C
D
A ~B
C D ~A
B ~C
```

3.4 Voorbeelduitvoer

```
4 3
A 1
B 1
C 0
D 1
A ~B
C D ~A
B ~C
```

4 Inleveren

Let op: deze opdracht wordt op een andere manier nagekeken dan de opdrachten van andere weken. Zo is het alleen implementeren van deze programma's slechts voldoende voor een 5; je dient optimalisaties te implementeren voor een hoger cijfer.

Hoe wordt de code nagekeken?

De punten voor deze opdracht worden (in principe) op de volgende manier verdeeld

- **2 punten** voor een correcte implementatie van de verifier
- **3 punten** voor een correcte implementatie van de solver (zonder optimalisaties)
- **3 punten** voor het implementeren van op zijn minst tweeserieuze optimalisaties en het beknopt beschrijven van deze implementaties in een apart document.
- **1 punt** voor de 40% snelste inzendingen
- **1 punt** voor de 20% snelste inzendingen

Optimalisaties

Een belangrijk onderdeel van de opdracht gaat om het implementeren van optimalisaties, slimmere manieren om het antwoord te vinden zonder alles te brute-forcen. Hierin is deze opdracht vrij open-ended, hoe je dit precies wil aanpakken mag je zelf bedenken. Je kunt in de eerste plaats uiteraard zelf zoeken naar slimmere manieren om bepaalde delen van de mogelijke toewijzingen over te slaan. Maar er zijn op internet ook veel optimalisaties en betere algoritmes te vinden om het satisfiabilityprobleem op te lossen. Je kunt de large-input gebruiken om te testen hoe lang je

code er ongeveer over doet. Voor het uiteindelijk bepalen van de snelste inzendingen wordt een andere inputfile gebruikt. Een goede start in het vinden van efficiëntere SAT-solvers is het hoofdstuk van SAT-solving in The Art of Computer Programming Volume 4, Fascicle 6 van Donald Knuth (link).

Nakijkeisen

- **De code dient geschreven te zijn in Python 3.**
- De code leest de input van *stdin* en schrijft naar *stdout*.
  Hint: de volgende voorbeeldcode leest op de juiste manier van stdin en schrijft hetzelfde naar stdout.

```
import fileinput
for line in fileinput.input():
    # line bevat al een newline van zichzelf
    print(line, end="")
```

- De verfifer moet de naam `ex6_v.py` hebben en de solver de naam `ex6_s.py`.
- Lever **alleen** de twee python bestanden met je code en een pdf met de beschrijvingen van je optimalisaties in. Lever dus geen zip/tar in of iets dergelijks.
- Je kunt de verifier gebruiken om je solver te controleren (je stuurt de output van de solver door naar de input van de verifier). Er is om die reden ook geen voorbeeldoutput bijgeleverd. Een manier om te testen:
  `cat small.input | python ex6_s.py | python ex6_v.py`

De code wordt hiernaast gecontroleerd op plagiaat van anderen en code op het internet. Hierop is de algemene plagiaat- en frauderegeling van de UvA van toepassing.

Geen inputfile ontvangen / overige vragen

Mocht je studentnummer niet tussen de inputfiles staan of mocht je overige vragen/problemen hebben over het inputformaat, stuur dan een mail naar zursebastian@gmail.com.