Rješavanje 3-sat problema

U propozicijskoj logici problem ispitivanja zadovoljivosti formule je problem pronalaženja modela formule: takve dodjele vrijednosti istinitosti logičkim varijablama da za njih formula postaje istinita. Primjerice, neka je zadana formula: $f_1(x_1, x_2, x_3, x_4) = \overline{x}_1 \cdot x_2 + x_3 \cdot x_4$. Ova formula je istinita za sve dodjele varijabli u kojima je $x_1=0$ i $x_2=1$ ili pak za sve dodjele varijabli u kojima je $x_3=1$ i $x_4=1$ (u formuli se množenje koristi kao kraći zapis operacije logičko-I a zbrajanje kao pokrata za logičko-ILI). Ako je formula zadana u obliku sume produkata, formula postaje istinita ako bilo koji njezin produkt postaje istinit, i to je tada trivijalno za utvrditi. Ako je formula zadana u obliku produkta suma, posao postaje značajno teži. Primjerice, formula:

$$f_2(x_1, x_2, x_3) = (x_1 + x_2 + x_3)(x_1 + x_2 + \overline{x}_3)(x_1 + \overline{x}_2 + x_3)(\overline{x}_1 + x_2 + x_3)(\overline{x}_1 + x_2 + \overline{x}_3)(\overline{x}_1 + \overline{x}_2 + \overline{x}_3)$$

ima samo dva modela: jedan je $x_1=0$, $x_2=1$, $x_3=1$ a drugi $x_1=1$, $x_2=1$, $x_3=0$, no to je teško vidljivo iz ovog zapisa. U zapisu produkta suma, svaku sumu ćemo zvati jednom klauzulom. Formula f_2 sastoji se od 6 klauzula. Svaka klauzula te formule sastoji se od tri literala, gdje je literal varijabla ili njezin komplement. Jedan od načina kako provjeriti je li formula zadana kao produkt suma zadovoljiva jest izmnožiti sve klauzule i tako formulu konvertirati u sumu produkata. Ako u tom zapisu postoji barem jedan produkt, formula je zadovoljiva i model možemo očitati direktno iz tog produkta; u suprotnom, formula je kontradikcija, nema modela i stoga nije zadovoljiva. Primjerice, množenjem svih klauzule formule f_2 dobili bismo:

$$f_2(x_1, x_2, x_3) = \overline{x}_1 \cdot x_2 \cdot x_3 + x_1 \cdot x_2 \cdot \overline{x}_3$$

iz kojega bismo mogli direktno očitati prethodno dana dva modela. Nažalost, svođenje formule iz produkta suma u sumu produkata nije izvedivo u polinomijalnom vremenu. Pogledajte to na primjeru funkcije f_2: da bismo izmnožili prve dvije klauzule, imamo $3x3=3^2$ množenja i potencijalno dobivamo upravo toliko različitih produkata. Potom to množimo sa sljedećom klauzulom koja u primjeru također ima 3 literala pa radimo $9x3=3^3$ produkata, i tako dalje. Za svođenje formule f_2 na produkt suma napravili bismo ukupno 3^6 množenja, odnosno općenito 3^m množenja gdje je *m* broj klauzula. Kako je potreban broj množenja eksponencijalan, postupak je i vremenski eksponencijalnog trajanja zbog čega je za iole složenije probleme neupotrebljiv. Alternativni algoritam mogli bismo formulirati na sljedeći način: u formuli f_2 pojavljuju se četiri varijable; stoga možemo krenuti u generiranje svih mogućih dodjela vrijednosti istinitosti za te četiri varijable i za svaku dodjelu možemo provjeriti postaje li njome formula istinita; ako postaje barem za jednu dodjelu, formula je zadovoljiva; u suprotnom smo dokazali da formula nije zadovoljiva. Nažalost, ovaj pokušaj neće se pokazati ništa upotrebljivijim: imamo li *n* varijabli, broj kombinacija koje trebamo ispitati je 2^n što je opet eksponencijalne vremenske složenosti.

Problem utvrđivanja zadovoljivosti formule koja je zadana kao produkt klauzula pri čemu se svaka klauzula sastoji od točno tri literala naziva se 3-*sat* problem. Dokazano je da je 3-*sat* problem *NP-potpun*.

Vaš je zadatak napisati program koji provjerava je li formula zadana u obliku produkta suma zadovoljiva. Program treba prihvaćati definiciju zadatka iz tekstovne datoteke. Format datoteke je sljedeći.

- Retci koji počinju znakom 'c' su komentari i mogu se preskočiti.
- Nailaskom retka koji počinje znakom '%' prestaje definicija problema i ostatak datoteke treba zanemariti.
- Od redaka nisu izbačeni sa prethodna dva pravila, prvi redak će sadržavati informacije o problemu a svi preostali retci definicije klauzula (jedna po retku).

Primjer definicije problema utvrđivanja zadovoljivosti formule f_2 prikazan je u nastavku. Formula je bila: $f_2(x_1, x_2, x_3)=(x_1+x_2+x_3)(x_1+x_2+\overline{x}_3)(x_1+\overline{x}_2+x_3)(\overline{x}_1+x_2+x_3)(\overline{x}_1+x_2+\overline{x}_3)(\overline{x}_1+\overline{x}_2+\overline{x}_3)$

```
c ovo je komentar...
c još malo komentara
p cnf 3 6
1 2 3 0
1 2 -3 0
1 -2 3 0
-1 2 3 0
-1 2 -3 0
-1 -2 -3 0
%
0
```

Brojevi u retku koji počinje s 'p' su broj varijabli (indeksiraju se od 1 do tog broja) te broj klauzula koje čine formulu. U ovom primjeru formula je definirana nad 3 varijable kroz 6 klauzula. Prikazani format za zapisivanje klauzula dozvoljava navođenje klauzula koje se sastoje od jednog ili više literala. Stoga se za svaku klauzulu navode indeksi varijabli nakon čega se popis terminira nulom. Ovakvo terminiranje uvedeno je kako bi se omogućilo navođenje više klauzula u jednom retku, no podatci s kojima ćete raditi uvijek će imati jednu klauzulu po retku. Indeksi varijabli će biti pozitivni, ako se odgovarajuća varijabla pojavljuje direktno odnosno negativni ako se varijabla u klauzuli pojavljuje komplementirana. Redak 1 2 3 0 stoga odgovara klauzuli:
$(x_1+x_2+x_3)$
dok redak 1 2 -3 0 odgovara klauzuli:
$(x_1+x_2+\overline{x}_3)$.

Napišite program koji se pokreće na sljedeći način:

```
java -cp staza hr.fer.zemris.trisat.TriSatSolver 1 uf20-010.cnf
```

pri čemu je prvi argument indeks željenog algoritma koji treba pokrenuti a drugi argument naziv (uz moguću putanju) do datoteke koja sadrži definiciju problema.

Rezultat pokretanja algoritma treba biti ili informacija da dokazivanje nije uspjelo jer su potrošeni resursi (primjerice, maksimalan broj iteracija) ili poruka da je formula zadovoljiva te koja je pronađena dodjela (zapišite je kao binarni vektor). Primjerice, za problem uf20-010.cnf, ako ga uspijete riješiti, ispis bi trebao biti:

```
Zadovoljivo: 00111101100101100010
```

(prvi bit s lijeve strane je vrijednost od x_1, sljedeći od x_2, ...)

Algoritam 1.

Ovaj algoritam ispituje sve moguće dodjele varijablama, ispisuje **SVA** pronađena rješenja, a pozivatelju vraća samo jedno od pronađenih rješenja (ili `null` ako nije pronađeno niti jedno rješenje). Ovo je algoritam iscrpnog pretraživanja i složenosti 2^m gdje je *m* broj varijabli. Ovim algoritmom provjerite radi li Vam učitavanje podataka za sve primjere te ispitivanje zadovoljava li dodjela klauzulu, formulu i slično.

Za potrebe debugiranja, evo rješenja koja bi ovaj algoritam morao pronaći za probleme s 20 varijabli.

Za uf20-01.cnf:
```
10000100100001101001
10000100000011101001
10010100000011101001
10000100100011101001
10010000010011101001
10010100010011101001
10010001010011101001
01110001111001101111
```

Za uf20-010.cnf:
```
00111101100001100010
00111101100101100010
00111101100101110010
10111100100111100011
00111101100111100011
10111101100111100011
10111100110111100011
01001111000111110011
00111101100111110011
```

Za uf20-0100.cnf:
```
01110011100111110010
01110011100111110011
01111011100111110011
01101101110011101011
```

Za uf20-01000.cnf:
```
01011010100000011010
```

Za probleme s 50 varijabli vjerojatno nećete imati vremena čekati na rezultat pa nemojte niti pokušavati s ovim algoritmom.

Za potrebe algoritama 2 i 3 problem ćemo formulirati kao optimizacijski problem.

Algoritam 2.

Pišete iterativni algoritam pretraživanja (predavanja, slide 26). Početnu dodjelu stvorite slučajno. Definirajte funkciju dobrote rješenja kao broj klauzula u formuli koje su tim rješenjem zadovoljene. Optimizacijski problem koji smo time definirali svodi se na pronalazak rješenja koje istovremeno zadovoljava što je moguće veći broj klauzula. Ako se pronađe rješenje koje zadovoljava sve klauzule, dokazali smo da je i formula zadovoljiva.
Definirajte funkciju pomaka $\pi(x)$ kao funkciju koja za rješenje x vraća sva rješenja koja se od rješenja x razlikuju u vrijednosti istinitosti jedne varijable. Primjerice, ako rješavamo problem od 4 varijable, i ako je trenutno najbolje rješenje 0011, tada je $\pi(0011)=\{1011, 0111, 0001, 0010\}$. Varijanta iterativnog algoritma koju trebate implementirati je pohlepni algoritam uspona na vrh (slide 27). Pretpostavimo da imate trenutno rješenje x dobrote *fit(x)*. Izračunajte njegovo susjedstvo te odredite njegov podskup koji čine rješenja s maksimalnom dobrotom. Taj skup ne mora nužno biti jednočlan. Ako je dobrota rješenja

u tom skupu manja od *fit(x)*, upali smo u lokalni optimum: prekinite program i dojavite neuspjeh. U suprotnom, kao sljedeće rješenje odaberite neko nasumično iz tog skupa. Postavite kao maksimalni broj iteracija vrijednost 100000.

Algoritam 3.

Algoritam 2 neće uvijek biti uspješan u rješavanju problema. Stoga ćemo pokušati napraviti modifikaciju koja će pametnije dodjeljivati dobrotu generiranom susjedstvu. Evo ideje. Svaki puta kada krenemo u novu iteraciju s rješenjem *x*, ažurirat ćemo statističke podatke o postotku iteracija u kojima je svaka klauzula bila zadovoljena. Ako imamo *m* klauzula, rezervirat ćemo polje od *m* vrijednosti tipa `double`: `post[1]` do `post[m]`. Inicijalno ćemo ove vrijednosti postaviti na `0`. Na početku svake iteracije uzet ćemo trenutno rješenje i provjeriti zadovoljava li ono svaku od klauzula. Razmotrimo sada *i*-tu klauzulu. Ako rješenje zadovoljava *i*-tu klauzulu, malo ćemo povećati vrijednost `post[i]`:
`post[i] += (1-post[i])*percentageConstantUp`
dok ćemo u slučaju da rješenje ne zadovoljava *i*-tu klauzulu malo smanjiti vrijednost `post[i]`:
`post[i] += (0-post[i])*percentageConstantDown`.
Na ovaj način kroz vrijeme računamo *procjenu* postotka iteracija u kojima je pojedina klauzula zadovoljena.

Nakon što ste ušli u iteraciju i ažurirali statističke podatke za klauzule, izračunajte susjedstvo trenutnog rješenja i svakog susjeda *x*' vrednujte na sljedeći način. Izračunajte broj klauzula koje *x*' zadovoljava. Označimo taj broj sa *Z*. Potom se spustite za razinu svake klauzule *i*, i vrijednosti Z dodajte korekciju klauzule koju ćete izračunati na sljedeći način:

- ako je klauzula *i* zadovoljena, korekcija iznosi:
  `percentageUnitAmount * (1-post[i])`
- a ako nije zadovoljena, korekcija iznosi:
  `-percentageUnitAmount * (1-post[i])`.

Uočite da ovako definiranim korekcijama, kad god klauzulu koju rijetko uspijevamo zadovoljiti (a te su najteže) zadovoljimo, dobroti dodajemo relativno velik pozitivan iznos (`post[i]` je pozitivan ali mali pa je `1-post[i]` blizu `1`). S druge pak strane, ako smo zadovoljili klauzulu koju i inače uspijevamo često zadovoljiti, iznos korekcije će biti blizak nuli. Također, ako imamo dodjelu koja ne zadovoljava klauzulu koju je i inače teško zadovoljiti, dodajemo veliku negativnu korekciju čime dosta rušimo dobrotu. Cilj ovakvog izračuna dobrote jest usmjeriti postupak pretraživanja prema prostoru u kojem uspijevamo zadovoljiti i najteže klauzule. Dobrota svakog susjeda sada će biti jednaka vrijednosti *Z* na koju su nadodane korekcije zbog svih klauzula.

Od svih susjeda pronađite njih `numberOfBest` najboljih, i potom iz tog popisa kao rješenje za sljedeću iteraciju uzmite neko nasumično odabrano.

Konstante neka budu sljedeće:

- `numberOfBest=2` (probajte i s drugim brojevima),
- `percentageConstantUp=0.01`,
- `percentageConstantDown=0.1` te
- `percentageUnitAmount=50`.

U ovoj izvedbi algoritma nemojte prekidati pretragu ako je novo rješenje gore od prethodnog rješenja.

Struktura rješenja

U nastavku su prikazani samo kosturi razreda, bez navođenja članskih varijabli. Funkcija metoda trebala bi biti jasna iz imena.

Definirajte razred `BitVector` koji nudi sljedeće "sučelje". Njegova zadaće jest predstavljati read-only dodjelu varijabli.

```
package hr.fer.zemris.trisat;

public class BitVector {

	public BitVector(Random rand, int numberOfBits) {...}

	public BitVector(boolean ... bits) {...}

	public BitVector(int n) {...}

	// vraća vrijednost index-te varijable
	public boolean get(int index) {...}

	// vraća broj varijabli koje predstavlja
	public int getSize() {...}

	@Override
	public String toString() {...}

	// vraća promjenjivu kopiju trenutnog rješenja
	public MutableBitVector copy() {...}
}
```

Definirajte razred `MutableBitVector` koji predstavlja verziju rješenja koja se može modificirati.

```
package hr.fer.zemris.trisat;

public class MutableBitVector extends BitVector {

	public MutableBitVector(boolean... bits) {...}

	public MutableBitVector(int n) {...}

	// zapisuje predanu vrijednost u zadanu varijablu
	public void set(int index, boolean value) {...}

}
```

Definirajte razred `Clause` koji predstavlja jednu klauzulu i nudi određene usluge.

```
package hr.fer.zemris.trisat;

public class Clause {

	public Clause(int[] indexes) {...}

	// vraća broj literala koji čine klauzulu
	public int getSize() {...}

	// vraća indeks varijable koja je index-ti član ove klauzule
	public int getLiteral(int index) {...}

	// vraća true ako predana dodjela zadovoljava ovu klauzulu
	public boolean isSatisfied(BitVector assignment) {...}

	@Override
	public String toString() {...}
}
```

Definirajte razred `SATFormula` koji predstavlja jednu formulu.

```
package hr.fer.zemris.trisat;

public class SATFormula {

	public SATFormula(int numberOfVariables, Clause[] clauses) {...}

	public int getNumberOfVariables() {...}

	public int getNumberOfClauses() {...}

	public Clause getClause(int index) {...}

	public boolean isSatisfied(BitVector assignment) {...}

	@Override
	public String toString() {...}
}
```

Definirajte razred `BitVectorNGenerator` koji predstavlja funkciju $\pi(x)$ i koji omogućava direktnu uporabu u for-petlji:

```
package hr.fer.zemris.trisat;

public class BitVectorNGenerator implements Iterable<MutableBitVector> {

	public BitVectorNGenerator(BitVector assignment) {...}

	// Vraća iterator koji na svaki next() računa sljedećeg susjeda
	@Override
	public Iterator<MutableBitVector> iterator() {...}

	// Vraća kompletno susjedstvo kao jedno polje
```

```
	public MutableBitVector[] createNeighborhood() {...}

}
```

Ovaj razred zamišljeno je da se može koristiti na sljedeći način:

```
BitVectorNGenerator gen = new BitVectorNGenerator(x0);
for(MutableBitVector n : gen) {
	// radi nešto sa susjedom
	...
}
```

Definirajte razred `SATFormulaStats` koji će Vam pomoći u dobivanju i ažuriranju svih podataka o nekom rješenju. Evo i njegove strukture:

```
package hr.fer.zemris.trisat;

public class SATFormulaStats {

	public SATFormulaStats(SATFormula formula) {...}

	// analizira se predano rješenje i pamte svi relevantni pokazatelji
	public void setAssignment(BitVector assignment, boolean updatePercentages) {...}

	// vraća temeljem onoga što je setAssignment zapamtio: broj klauzula koje su zadovoljene
	public int getNumberOfSatisfied() {...}

	// vraća temeljem onoga što je setAssignment zapamtio
	public boolean isSatisfied() {...}

	// vraća temeljem onoga što je setAssignment zapamtio: suma korekcija klauzula
	public double getPercentageBonus() {...}

	// vraća temeljem onoga što je setAssignment zapamtio: procjena postotka za klauzulu
	public double getPercentage(int index) {...}
}
```

Konačno, napišite i razred `TriSATSolver` koji sadrži metodu main. Razmislite kako najbolje napraviti modeliranje prikladne strukture algoritama te izvedbu parsera.

Na Ferka ćete trebati uploadati ZIP arhivu s Eclipse projektom rješenja. Rok: ponedjeljak, 11. listopada u 08:00 ujutro.