Univerzitet u Beogradu
Elektrotehnički fakultet

Veštačka inteligencija

SAT problem i rešavanje problema svođenjem na SAT problem

Jovana Savić 2020/3423

mentor
Prof. dr Tatjana Lutovac

Glava 1

Uvod

SAT problem (*eng.* satisfiability problem) je jedan od centralnih problema koji se javlja u iskaznoj logici. Ovaj problem podrazumeva ispitivanje zadovoljivosti formule, odnosno, određivanje valuacije u kojoj je formula tačna. SAT problem je predstavnik NP-kompletnih problema i prvi problem za koji je dokazano da je NP-kompletan [1].

U poglavlju 2 je dat pregled osnovnih aspekata iskazne logike. Definisane su sintaksa i semantka iskazne logike, kao i logičke posledice i logički ekvivalentne formule.

U poglavlju 3 su objašnjene konjuktivne i disjunktivne normalne forme. Dat je algoritam uz pomoć kog se bilo koja logička formula može svesti na KNF oblik. U ovom poglavlju je definisan SAT problem i data procedura za njegovo rešavanje - DPLL algoritam, koji je trenutno najbolji algoritam na raspolaganju i predstavlja osnovu modernih SAT rešavača. Objašnjeno je šta su to SAT rešavači, dat je pregled mogućnosti koje oni poseduju i kratak pregled tehnika kojima se unapređuje osnovni DPLL algoritam.

U poglavlju 4 su dati neki primeri svođenja na SAT problem. Ovi primeri pokazuju kako se neki problem može svesti na SAT problem. Pokazano je kako se jednostavni problemi mogu prilagoditi SAT rešavačima koji očekuju funkciju u KNF obliku, ali je pokazano i kako se rešavaju kompleksniji problemi (u kojima nije tako jednostavno problem direktno definisati u KNF obliku) korišćenjem `z3py` biblioteke.

Glava 2

Iskazna logika

U ovoj glavi je dat pregled osnovnih pojmova iz iskazne logike koji je uglavnom preuzet iz [2] i [3]. U iskaznoj logici (*eng.* propositional logic) promenljive predstavljaju iskaze, odnosno, tvrđenja. Iskazi se korišćenjem logičkih veznika mogu kombinovati u složenije logičke iskaze. Iskazna logika ima tri aspekta: svoju sintaksu (jezik), semantiku (značenje iskaza) i svoje deduktivne sisteme. Centralni problemi u iskaznoj logici su ispitivanje da li je formula valjana, odnosno tautologija (*eng.* tautology) i da li je formula zadovoljiva (*eng.* satisfiable). Ispitivanje zadovoljivosti formule podrazumeva određivanje valuacije u kojoj je formula tačna, i ovaj problem je poznat kao SAT problem (*eng.* satisfiability problem).

2.1 Sintaksa iskazne logike

Definicija 2.1.1. Neka je alfabet Σ unija sledeća četiri skupa:

1. prebrojivog skupa iskaznih slova P;
2. Skupa logičkih veznika $\{\neg, \vee, \wedge, \Rightarrow, \Leftrightarrow\}$, pri čemu je $\neg$ unarni veznik, a $\vee, \wedge, \Rightarrow$ i $\Leftrightarrow$ su binarni veznici;
3. Skupa logičkih konstanti $\{\top, \bot\}$;
4. Skupa pomoćnih simbola $\{(,)\}$.

Skup iskaznih formula (ili jezik iskazne logike) nad skupom P je najmanji podskup skupa svih reči nad Σ takav da važi:

- iskazna slova (iz skupa P) i logičke konstante su iskazne formule
- ako su A i B iskazne formule onda su iskazne formule i objekti dobijeni kombinovanjem ovih formula logičkim veznicima.

Definicija 2.1.2. Rezultat zamene (supstitucije) svih pojavljivanja iskazne formule C u iskaznoj formuli A iskaznom formulom D označavamo sa $A\,[C \to D]$. Ta zamena se definiše na sledeći način:

- ako za iskazne formule A i C važi $A = C$, onda je $A\,[C \to D]$ jednako D;

- ako za iskazne formule A i C važi $A \neq C$ i A je atomička iskazna formula, onda je $A\,[C \to D]$ jednako A;
- ako za iskazne formule A, B, i C važi $A \neq C$ i $A = \neg B$, onda je $A\,[C \to D] = \neg\,(B\,[C \to D])$;
- ako za iskazne formule A, B_1, B_2 i C važi $A \neq C$ i formula A jednaka je $(B_1 \vee B_2)$, $(B_1 \wedge B_2)$, $(B_1 \Rightarrow B_2)$ ili $(B_1 \Leftrightarrow B_2)$, onda je formula $A\,[C \to D]$ jednaka redom $(B_1\,[C \to D] \vee B_2\,[C \to D])$, $(B_1\,[C \to D] \wedge B_2\,[C \to D])$, $(B_1\,[C \to D] \Rightarrow B_2\,[C \to D])$ i $(B_1\,[C \to D] \Leftrightarrow B_2\,[C \to D])$.

2.2 Semantika iskazne logike

Semantički aspekt iskazne logike govori o značenju formula.

Definicija 2.2.1. Interpretacija I_v za valuaciju v definiše se na sledeći način:

- $I_v(\top) = 0$ i $I_v(\bot) = 1$;
- $I_v(p) = v(p)$ za svaki element p skupa P;
- $I_v(\neg A) = \begin{cases} 1, & \text{ako je } I_v(A) = 0 \\ 0, & \text{inače} \end{cases}$
- $I_v(A \vee B) = \begin{cases} 0, & \text{ako je } I_v(A) = 0 \text{ i } I_v(B) = 0 \\ 1, & \text{inače} \end{cases}$
- $I_v(A \wedge B) = \begin{cases} 1, & \text{ako je } I_v(A) = 1 \text{ i } I_v(B) = 1 \\ 0, & \text{inače} \end{cases}$
- $I_v(A \Rightarrow B) = \begin{cases} 0, & \text{ako je } I_v(A) = 1 \text{ i } I_v(B) = 0 \\ 1, & \text{inače} \end{cases}$
- $I_v(A \Leftrightarrow B) = \begin{cases} 1, & \text{ako je } I_v(A) = I_v(B) \\ 0, & \text{inače} \end{cases}$

Definicija 2.2.2. Valuacija v je zadovoljavajuća za formulu A ako je $I_v(A) = 1$. Kažemo i da je zadovoljavajuća valuacija v za model A i pišemo $v \models A$.

Definicija 2.2.3. Iskazna formula A je:

- zadovoljiva (eng. *satisfiable*) ako postoji valuacija koja je za nju zadovoljiva;
- valjana ili tautologija (eng. *valid, tautology*) ako je svaka valuacija za nju zadovoljavajuća, to jest, ako za svaku valuaciju v važi $v \models A$, i to zapisujemo $\models A$;
- nazadovoljiva ili kontradikcija (eng. *unsatisfiable, contradictory*) ukoliko ne postoji valuacija koja je za nju zadovoljavajuća;
- poreciva (eng. *falsifiable*) ako postoji valuacija koja za nju nije zadovoljavajuća.

Definicija 2.2.4. Skup iskaznih formula Γ je:

- zadovoljiv, ako postoji valuacija v u kojoj je svaka formula iz Γ tačna. Za takvu valuaciju v se kaže da je model za Γ;
- nezadovoljiv ili kontradiktoran, ako ne postoji valuacija u kojoj je svaka formula iz Γ tačna.

Naredna teorema direktno sledi iz prethodne definicije.

Teorema 2.2.1. Valuacija v je model skupa formula $\{A_0, A_1, \ldots, A_n\}$ ako i samo ako je v model formule $A_0 \wedge A_1 \wedge \cdots \wedge A_n$.

2.3 Logičke posledice i logički ekvivalentne formule

Definicija 2.3.1. (*Logička posledica i logička ekvivalnecija*) Ako je svaki model za skup iskaznih formula Γ istovremeno i model za iskaznu formulu A, onda se kaže da je A logička posledica (eng. *logical consequence*) skupa Γ i piše se $\Gamma \models A$.

Ako je svaki model iskazne formule A i iskazne formule B i obratno (tj. ako važi $\{A\} \models B$ i $\{B\} \models A$), onda se kaže da su formule A i B logički ekvivalentne (eng. *logically equivalent*) i piše se $A \equiv B$.

Teorema 2.3.1. Važi $A_1, \ldots, A_n \models B$ ako i samo ako važi $A_1 \wedge \ldots \wedge A_n \models B$.

Teorema 2.3.2. Važi $A \models B$ ako i samo ako je iskazna formula $A \Rightarrow B$ tautologija.

Važi $A \equiv B$ ako i samo ako je iskazna formula $A \Leftrightarrow B$ tautologija.

Teorema 2.3.3. *(Teorema o zameni)* Ako je $C \equiv D$ onda je $A\,[C \to D] \equiv A$.

Glava 3

Ispitivanje zadovoljivosti u iskaznoj logici

U praktičnim primenama, onda kada je problem formulisan na jeziku iskazne logike, centralni problem postaje ispitivanje da li je neka iskazna formula tautologija, da li je zadovoljiva, poreciva ili kontradikcija. Ova četiri problema su povezana, i svaka tri se mogu svesti na četvrti. Fokus je uglavnom na ispitivanju zadovoljivosti neke formule A, a potom se isti algoritam može primeniti na rešavanje ostala tri problema. Formula A je tautologija ako $\neg A$ nije zadovoljiva, A je poreciva ako je $\neg A$ zadovoljiva, i konačno, A je kontradikcija ako A nije zadovoljiva.

Prema tome, dovoljno je pronaći efikasan algoritam za rešavanje problema zadovoljivosti, a ostale vrste problema svesti na ovaj na prethodno opisan način.

3.1 Normalne forme i potpuni skupovi veznika

Definicija 3.1.1. *(Konjuktivna normalna forma)* Iskazna formula je u konjuktivnoj normalnoj formi (KNF) ako je oblika

$$A_1 \wedge A_2 \wedge \ \ldots \ \wedge A_n$$

pri čemu je svaka od formula A_i $(1 \leq i \leq n)$ klauza (tj. disjnukcija literala).

Definicija 3.1.2. *(Disjnuktivna normalna forma)* Iskazna formula je u disjunktivnoj normalnoj formi (DNF) ako je oblika

$$A_1 \vee A_2 \vee \ \ldots \ \vee \ A_n$$

pri čemu je svaka od formula A_i $(1 \leq i \leq n)$ konjukcija literala.

Korišćenjem pogodnih ekvivalencija, svaka iskazna formula može biti transformisana u svoju konjuktivnu, odnosno disjunktivnu, normalnu formu. Transformisanje iskazne formule u konjuktivnu normalnu formu je opisano KNF algoritmom koji navodimo u nastavku.

Teorema 3.1.1. *(Korektnost algoritma KNF)* Algoritam KNF se zaustavlja i zadovoljava sledeće svojstvo: ako je F ulazna formula, onda je izlazna formula u konjuktivnoj normalnoj formi i logički je ekvivalentna sa F.

Dokaz ove teoreme se može naći u [3].

Algoritam 1: KNF algoritam

Input: Iskazna formula F
Output: Konjuktivna normalna forma formule F
while *postoji veznik* $\Leftrightarrow$ **do**
  `// eliminiši veznik primenom logičke ekvivalencije`
  $A \Leftrightarrow B \equiv (A \Rightarrow B) \wedge (B \Rightarrow A)$;
end
while *postoji veznik* $\Rightarrow$ **do**
  `// eliminiši veznik primenom logičke ekvivalencije`
  $A \Rightarrow B \equiv \neg A \vee B$;
end
while *moguće* **do**
  `// primeni neku od logičkih ekvivalencija`
  $\neg(A \wedge B) \equiv \neg A \vee \neg B$;
  $\neg(A \vee B) \equiv \neg A \wedge \neg B$;
end
while *postoje višestruki veznici* **do**
  `// eliminiši višestruke veznike primenom logičke ekvivalencije`
  $\neg\neg A \equiv A$;
end
while *moguće* **do**
  `// primeni neku od logičkih ekvivalencija`
  $A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$;
  $(B \wedge C) \vee A \equiv (B \vee A) \wedge (C \vee A)$;
end

3.2 Problem SAT i DPLL procedura

Problem ispitivanja zadovoljivosti date iskazne formule u KNF obliku označava se SAT (*eng.* satisfiability problem - problem zadovoljivosti). SAT problem je NP-kompletan i ima ogroman teoretski i praktični značaj.

Kako se još uvek ne zna da li su klase P i NP problema jednake, još uvek se ne zna ni da li postoji algoritam polinomske složenosti za rešavanje SAT problema. Problem ispitivanja zadovoljivosti formula koje su date u DNF obliku je očigledno trivijalan obzirom da je dovoljno razmatrati disjunkte pojedinačno i pripada klasi P. Međutim, svođenje SAT problema na problem ispitivanja zadovoljivosti DNF formule nije, u opštem slučaju, razuman put zbog kompleksnosti same transformacije.

Dejvis-Patnam-Logman-Lavlendova ili DPLL procedura je procedura za ispitivanje zadovoljivosti iskaznih formula u KNF obliku, to jest, procedura za rešavanje instanci SAT problema. U proceduri se podrazumeva da je prazan skup klauza zadovoljiv i da je klauza koja ne sadrži nijedan literal (prazna klauza) nezadovoljiva i formula koja sadrži praznu klauzu je nezadovoljiva. Osnovni oblik ovog algoritma je dat na strani 8.

Teorema 3.2.1. *(Korektnost DPLL procedure)* Za svaku iskaznu formulu DPLL procedura se zaustavlja i vraća odgovor DA ako i samo ako je polazna formula zadovoljiva.

Dokaz ove teoreme se može naći u [3].

Procedura DPLL je u najgorem slučaju eksponencijalne vremenske složenosti po broju iskaznih promenljivih u formuli, usled rekurzivne primene pravila *split*. Ova procedura se može smatrati algoritmom pretrage potpunog stabla valuacija promenljivih koje učestvuju u formuli. Koraci algoritma omogućuju da se ne pretražuje nužno čitavo stablo. Postoje razne heuristike koje značajno mogu da ubrzaju algoritam tako što usmeravaju pretragu. Neke od najčešće korišćenih su date u narednom odeljku.

Algoritam 2: DPLL algoritam

Input: Multiskup klauza D ($D = \{C_1, C_2, \ldots, C_n\}$)
Output: DA, ako je multiskup D zadovoljiv, NE, inače
if *D je prazan* **then**
 | **return** *DA*
end
zameni sve literale $\neg\bot$ sa $\top$ i zameni sve literale $\neg\top$ sa $\bot$;
obriši sve literale jednake $\bot$;
if *D sadrži praznu klauzu* **then**
 | **return** *DA*
end
// Korak *tautology*
if *neka klauza C_i sadrži $\top$ ili sadrži neki literal i njegovu negaciju* **then**
 | **return** *DPLL($D \backslash C_i$)*
end
// Korak *unit propagation*
if *neka klauza je jedinična i jednaka nekom iskaznom slovu p* **then**
 | **return** *DPLL($D\,[p \mapsto \top]$)*
end
if *neka klauza je jedinična i jednaka $\neg p$ za neko iskazno slovo p* **then**
 | **return** *DPLL($D\,[p \mapsto \bot]$)*
end
// Korak *pure literal*
if *D sadrži literal p (gde je p neko iskazno slovo), ali ne i $\neg p$* **then**
 | **return** *DPLL($D\,[p \mapsto \top]$)*
end
if *D sadrži literal $\neg p$ (gde je p neko iskazno slovo), ali ne i p* **then**
 | **return** *DPLL($D\,[p \mapsto \bot]$)*
end
// Korak *split*
if *DPLL($D\,[p \mapsto \top]$) (gde je p jedno od iskaznih slova koja se javljaju u D) = DA* **then**
 | **return** *DA*
end
return *DPLL($D\,[p \mapsto \bot]$)*

3.3 SAT rešavači

Programi koji su specijalizovani za rešavanje SAT problema se zovu SAT rešavači i u poslednje dve decenije su doživeli ogroman napredak. Od 2002. godine održava Internacionalno takmičenje u rešavanju SAT problema koje je dovelo do razvoja brojnih pametnih implementacija i tehnika. Kao što je već pomenuto, ne postoji algoritam za rešavanje SAT problema koji u najgorem slučaju nema eksponencijalnu složenost, međutim, moderni SAT rešavači su u stanju da reše probleme koji imaju i preko milion promenljivih i nekoliko miliona ograničenja [4].

Svođenje problema na SAT problem uglavnom dovodi do povećanja prostora pretrage problema, ali i pored toga, danas je često mnogo jednostavnije problem svesti na SAT i potom iskoristiti jedan od modernih SAT rešavača[1], nego osmisliti algoritam koji je prilagođen konkretnom problemu.

Takođe, ispostavlja se da je većina slučajno generisanih SAT problema prilično jednostavna za rešavanje, dok samo mali procenat predstavlja najgori slučaj koji je eksponencijalne složenosti. Međutim, moderni SAT rešavači koji koriste lokalne tehnike pretrage uspevaju da reše i takve probleme. Uspeh koji su postigli SAT rešavači je zbog toga otvorio pitanje da li je ipak ima više smisla razmatrati složenost algoritama u tipičnom umesto u najgorem slučaju.

SAT rešavači se mogu podeliti na dve osnovne grupe - kompletni i nekompletni SAT rešavači. Kompletni SAT rešavači su oni koji za datu iskaznu formulu vraćaju valuaciju za koju je ona zadovoljiva, odnosno, potvrđuju da je formula kontradikcija. Ovi rešavači se baziraju na DPLL proceduri i unapređuju je uvođenjem raznih heuristika. Nekompletni SAT rešavači ne garantuju da će algoritam proizvesti rezultat, pa se uglavnom uvode dodatna ograničenja na osnovu kojih se algoritam zaustavlja ukoliko nije došao do rezultata. Ovi rešavači se baziraju na stohastičkoj lokalnoj pretrazi i za mnoge klase problema daju mnogo bolje rezultate nego kompletni SAT rešavači.

U nastavku navodimo nekoliko osnovnih tehnika koje su doprinele poboljšanju kompletnih SAT rešavača.

- *Izbor iskaznog slova u koraku split* ili *strategija odlučivanja* je jedna od osnovnih tehnika za unapređivanje DPLL algoritma i ima jako veliki uticaj na efikasnost algoritma. Neke tehnike podrazumevaju slučajno biranje promenljivih, dok druge mogu birati ona iskazna slova koja se najčešće pojavljuju u tekućoj formuli, ili ona koja koja se pojavljuju u najkraćim klauzama i slično.

- *Učenje klauza* je tehnika koja je odigrala veoma značajnu ulogu u unapređivanju SAT rešavača. Ideja je da se izvori konflikta čuvaju tako da se kasnije iskoriste za smanjenje prostora pretrage.

- *Praćenje iskaznih slova* (*eng.* Watched literals scheme) [5] je struktura podataka koja omogućava efikasnije izvršavanje DPLL algoritma. U ovoj implementaciji svaka klauza posmatra dva iskazna slova koja nisu $\perp$ osim ako klauza nije zadovoljena. Svako iskazno slovo ima listu klauza koje ga posmatraju. Kada se iskaznom slovu dodeli neka vrednost, ukoliko se time prekrši pravilo, klauza se vezuje za sledeće iskazno slovo. Ukoliko ono ne postoji klauza je jedinična i ide se na *unit propagation* korak.

[1] Većina SAT rešavača se može potpuno besplatno preuzeti sa interneta.

- *Skok nazad na osnovu konflikta* (*eng.* conflict-based backjumping), prvi put uvedena u [6], je tehnika koja podrazumeva da se pretraga vrati na onaj nivo na kome je određena vrednost promenljive koja je na nižim nivoima dovela do konflikta.

- *Minimizacija konfliktnih klauza* (*eng.* conflict clause minimization) je ideja slična prethodnoj. I ovde je cilj dobiti što kraće konfliktne klaze. Kada se kreira konfliktna klauza ona pokušava da se skrati tako što se izbacuju promenljive koje će biti netačne ukoliko su ostale promenljive u klauzi netačne [7]

- *Slučajni restart* (*eng.* randomized restarts) je tehnika koja je prvi put uvedena u [8]. Ideja je da se nakon određenog vremena algoritam zaustavi i ponovo počne. Sve konfliktne klauze koje su u međuvremenu pronađene se dodaju originalnoj formuli.

Klasični SAT rešavači obično očekuju ulaz u DIMACS-CNF formatu. U ovom formatu prvi red sadrži broj iskaznih promenljivih i broj klauza. Potom se u svakom redu daje zapis klauze. U klauzi su promenljive označene svojim rednim brojevima, a negirane promenljive se predstavljaju negativnim brojevima.

Osim klasičnih SAT rešavača, u ovom radu je korišćen i `z3py` [9]. Radi se o biblioteci za programski jezik `Python` za automatsko dokazivanje teorema. Ova biblioteka ima ugrađen SAT rešavač i omogućava mnogo lakše definisanje problema. Korisnik ne mora da kreira KNF formulu koja mu je potrebna, već može da se fokusira na jasan opis problema, a sama biblioteka obavlja posao svođenja formula na odgovarajuće oblike.

Glava 4

Svođenje na SAT problem

Mnogi problemi koji se rešavaju nad konačnim domenima se mogu pogodno opisati svođenjem na SAT problem. U [2] se može videti kako se problemi poput problema n dama, igre čistač mina, raspoređivanja sportskih utakmica, provere ekvivalentnosti kombinatornih kola, verifikacije softvera i ostalih mogu svesti na SAT problem. U [9] se može pronaći rešavanje sudokua i problema instalacije korišćenjem SAT rešavača u programskom jeziku Python. U [10] je pokazano kako se mogu rešavati CSP problemi svođenjem na SAT, u master radu [11] se kriptografksi problemi rešavaju svođenjem na SAT problem, dok se u [12], igre parnosti rešavaju svođenjem na SAT.

Veliki napredak koji su ostvarili SAT rešavači i biblioteke poput `z3py` koje omogućavaju apstraktije formulacije problema su glavni argumenti za pristup koji prodrazumeva rešavanje problema svođenjem na SAT umesto nekog posebno prilagođenog algoritma.

Prvo se, uglavnom, elementarni iskazi koji opisuju problem opišu iskaznim promenljivima. Potom se uslovi problema predstave iskaznim formulama nad tim iskaznim promenljivim. Konjukcija iskaznih formula se nakon toga transformiše u KNF oblik i potom se nekim od SAT rešavača ispituje zadovoljivost, odnosno nalazi model koji daje rešenje polaznog problema.

4.1 Kodiranje

U većini praktičnih problema se srećemo promenljive koje nisu binarne, već celobrojne, ali uzimaju vrednosti iz nekog ograničenog skupa. U takvim situacijama se koristi kodiranje. Postoje dva osnovna kodiranja - retka kodiranja (*eng.* sparse encoding) i log kodiranja (*eng.* log encoding).

U slučaju retkog kodiranja uvodimo iskazne promenljive $p_{v,i}$ koje su tačne ako i samo ako promenljiva v ima vrednost i. Tada se uslov da promenljiva v ima vrednost iz domena I obezbeđuje formulom

$$\bigvee_{i \in I} p_{v,i}.$$

Sa druge strane, promenljiva ne sme imati više od jedne vrednosti i taj uslov zapisujemo formulom

$$\bigwedge_{i,j \in I, i \neq j} \neg p_{v,i} \vee \neg p_{v,j}.$$

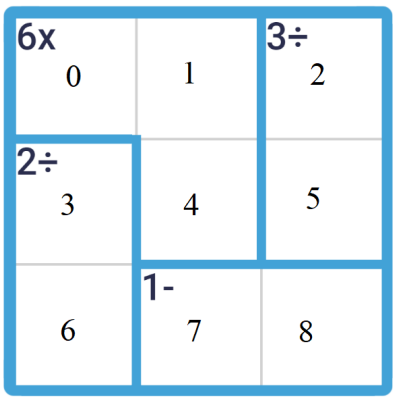
Ove dve formule su potrebne da bismo iskazali da promenljiva ima tačno jednu vrednost iz nekog konačnog skupa I.

U log kodiranju svakom bitu vrednosti numeričke promenljive (zapisanoj u binarnoj reprezentaciji) pridružujemo jednu iskaznu promenljivu. Notacija koja se koristi za iskazne promenljive je $p_{v,i}$ gde v označava promenljivu, a i označava bit. Takođe, podrazumeva se da koristimo najmanji broj bitova koji je potreban za predstavljanje svih vrednosti koje promenljiva može da ima. Naravno, u ovom slučaju nije potrebno definisati gore date uslove koji obezbeđuju jednu i samo jednu vrednost. Sa druge strane, ukoliko promenljiva ne može imati vrednosti koje se mogu prikazati iskaznim promenljivim, onda je potrebno zabraniti ih dodatnim klauzama (npr. ako promenljiva v može imati vrednosti iz skupa $\{0, \ldots, 6\}$, tada treba dodati uslov $\bigvee_{i=0,1,2} \neg p_{v,i}$).

Očigledno, ono što je moguće predstaviti retkim kodiranjem možemo predstaviti i log kodiranjem i obratno. Izbor kodiranja najviše zavisi od samog problema i od toga šta nam omogućava da jasnije modelujemo problem.

4.2 Igra *mathdoku*

U ovom odeljku ćemo na primeru igre *mathdoku* ilustrovati čisto svođenje na SAT problem, to jest, svođenje na problem tako da se on direktno može proslediti SAT rešavaču u DIMACS-CNF formatu. U ovoj igri se koriste table raznih veličina, ali su osnovna pravila ista. Ako je tabla dimenzija $n \times n$ tada se upisuju brojevi od 1 do n. Slično kao kod igre sudoku, u svakom redu i u svakoj koloni se mora pojaviti svaki broj tačno jednom. Tabla je podeljena na okvire i za svaki je dat broj i operacija - brojevi u okviru korišćenjem te operacije treba da daju definisani rezultat. Na slici pored je prikazan primer najmanje table na kojoj ćemo ilustrovati svođenje na SAT problem.



Promenljive u našem zadatku su brojevi koje treba upisati i u ovom slučaju ćemo koristiti retko kodiranje. Promenljive ćemo označiti brojevima od 0 do 8 kao što je prikazano na slici i ovaj skup ćemo označiti sa D. Promenljive uzimaju vrednosti iz domena $I = \{1, 2, 3\}$.

Prvo je potrebno obezbediti uslove da svaka promenljiva uzima vrednost iz domena

$$\bigwedge_{v \in D} (p_{v,1} \vee p_{v,2} \vee p_{v,3}) \tag{4.1}$$

i da promenljiva ne može imati dve vrednosti istovremeno

$$\bigwedge_{v\in D}\bigwedge_{i,j\in I, i\neq j}\left(\neg p_{v,i}\vee\neg p_{v,j}\right). \tag{4.2}$$

Potom uvodimo ograničenja koja predstavljaju pravila igre. Prvo ćemo razmatrati ograničenja koja govore o tome da se u svakom redu i u svakoj koloni pojavljuje svaki broj tačno jednom. Ako kolone i redove numerišemo brojevima 0, 1, 2, tada se promenljiva p_v nalazi u $v/3$ redu i v mod 3 koloni, gde je sa / označeno celobrojno deljenje.

Ako promenljiva p_v ima vrednost i tada sve promenljive koje su u istom redu i koloni treba da imaju vrednosti koje su različite od i. Ovo možemo da zapišemo kao

$$\bigwedge_{v\in D}\bigwedge_{i\in I}\bigwedge_{w\in W}\left(\neg p_{v,i}\vee\bigvee_{j\in I/\{i\}}p_{w,j}\right) \tag{4.3}$$

gde $W=\{w\in D\mid w/3=v/3\vee w\equiv v \bmod 3\}/\{v\}$.

Konačno, definisaćemo uslove koji govore o operacijama unutar okvira.

Posmatrajmo okvir koji čiji je uslov $p_0\cdot p_1\cdot p_4=6$. Jasno je da će ovaj uslov biti ispunjen ako su p_0,p_1,p_4 iz skupa $\{1,2,3\}$ i imaju različite vrednosti. Prema tome, ovaj uslov je sličan onome koji važi za redove i kolone:

$$\bigwedge_{v\in B_1}\bigwedge_{i\in I}\bigwedge_{w\in B_1, w\neq v}\left(\neg p_{v,i}\vee\bigvee_{j\in I/\{i\}}p_{w,j}\right) \tag{4.4}$$

gde je $B_1=\{0,1,4\}$.

Posmatrajmo dalje okvir u kome treba da važi $p_3/p_6=2$ ili $p_6/p_3=2$. Ovaj uslov će biti ispunjen ako je $(p_3,p_6)\in\{(1,2),(2,1)\}$.

$$\begin{aligned}&(p_{3,1}\wedge p_{6,2})\vee(p_{3,2}\wedge p_{6,1})\\&=(p_{3,1}\vee p_{3,2})\wedge(p_{3,2}\vee p_{6,2})\wedge(p_{3,1}\vee p_{6,1})\wedge(p_{6,1}\vee p_{6,2})\end{aligned} \tag{4.5}$$

Slično važi i za okvir u gornjem desnom uglu u kome važi $p_2/p_5=3$ ili $p_5/p_2=3$. U ovom slučaju dodajemo uslov

$$(p_{2,1}\vee p_{2,3})\wedge(p_{2,3}\vee p_{5,3})\wedge(p_{2,1}\vee p_{5,1})\wedge(p_{5,1}\vee p_{5,3}) \tag{4.6}$$

Poslednji uslov je da razlika brojeva u poljima 7 i 8 bude jednaka jedinici. Očigledno, ovo znači da bar jedan od ta dva broja mora biti broj 2, a drugi mora onda biti 1 ili 3. Ovaj uslov predstavljamo sledećom formulom

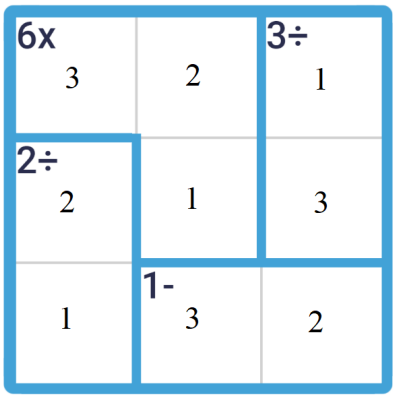
$$(\neg p_{7,2}\vee p_{8,1}\vee p_{8,3})\wedge(\neg p_{8,2}\vee p_{7,1}\vee p_{7,3})\wedge(p_{7,2}\vee p_{8,2}) \tag{4.7}$$

Ovaj primer je rešen uz pomoć programa `cryptominisat` koji omogućava rešavanje SAT problema u pretraživaču bez instalacija[1]. Naravno, ovakvi rešavači su ograničeniji, ali su za jednostavne probleme poput ovog sasvim zadovoljavajući. Za generisanje oblika formula koji se može proslediti SAT rešavaču je korišćen jezik Python. Dobijeno rešenje je prikazano na slici i vidimo da je ono korektno.

[1] `https://msoos.github.io/cryptominisat_web/`

Na ovom primeru smo ilustrovali kako se jedan problem može svesti na SAT problem, i dato rešenje se lako može modifikovati i primneiti na klasičan sudoku ili još neke modifikacije ove igre. U ovom slučaju generisanje formule u KNF obliku nije bilo teško, ali, u opštem slučaju, direktna formulacija problema u KNF obliku jeste jako komplikovano.

Programski kôd i rezultat programa su dati u prilogu A.



4.3 Igra podmornica - lakša verzija igre

Na slici 4.1 je prikazana igra podmornica za jednog igrača[2]. Brojevi sa desne strane označavaju broj popunjenih polja u datom redu, dok brojevi koji se nalaze ispod označavaju broj popunjenih polja u svakoj koloni. Sa strane su prikazane podmornice koje su skrivene, i njihov broj zavisi od veličine table. Najmanja zauzima jedno polje, a najveća četiri polja. Polja koja predstavljaju jednu podmornicu su sa svih strana ograničena vodom, odnosno, sigurno su prazna (uključujući i ona koja su povezana dijagonalno). Kada se započne igra neka polja su već popunjena. Igra je takva da rešenje postoji i jednistveno je. U lakšoj verziji ove igre se na osnovu toga koliko je polja popunjeno i na osnovu pravila da se brodovi ne smeju dodirivati moguće popuniti tablu. Nije neophodno praćenje toga da li je broj različitih podmornica zadovoljen.

Za ovaj problem ćemo koristiti `Python z3 solver`. Na ovom primeru ćemo ilustrovati kako se na SAT svode problemi koji su kompleksniji i za koje nije baš jednostavno precizno formulisati formulu u KNF obliku. Za razliku od SAT rešavača koji je korišćen u prethodnom primeru, ovaj rešavač je dosta apstraktniji i omogućava mnogo jednostavnije modelovanje problema, bez detalja koji su neophodni za klasičan SAT rešavač.

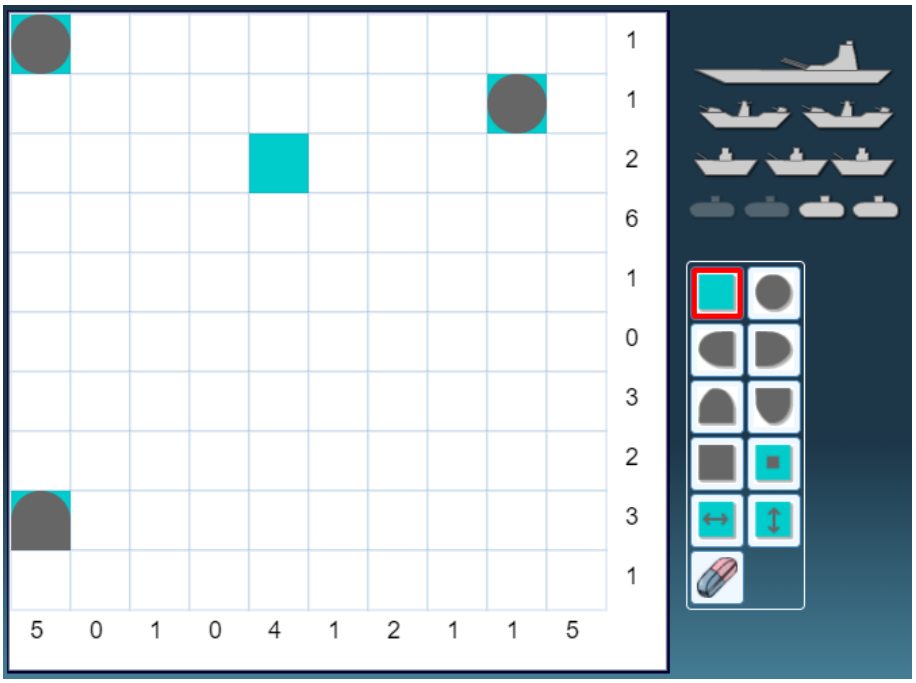
Ovaj problem ćemo modelovati uvođenjem promenljivih $p_{i,j}$, $0 \leq i, j < 10$, koje predstavljaju polja i imaju vrednost 1 ukoliko se na datom polju nalazi podmornica. Smatraćemo da je polje $p_{0,0}$ u gornjem levom uglu.

Prvo ćemo razmatrati uslove koji se tiču broja popunjenih polja u redovima i kolonama. Sa r_i ćemo označiti broj popunjenih polja u i-tom redu, a sa c_j broj popunjenih polja u koloni j. Smatrajući da naše promenljive uzimaju vrednosti 0 i 1 ovi uslovi se mogu pisati kao:

$$\sum_{i=0}^{9} p_{i,j} = c_j, \quad \sum_{j=0}^{9} p_{i,j} = r_i \tag{4.8}$$

Ovakvi uslovi se često pojavljuju kada pokušavamo da svedemo probleme na SAT i u pitanju su pseudo-bulova ograničenja (*eng.* pseudo-boolean constraints). Predstavljanje ovakvih ograničenja tako da se positgne što veća efikasnost kada

[2]Igra se može naći na sajtu `https://dkmgames.com/Yubotu/`



Slika 4.1: Igra podmornica - lakša verzija

se ista ubace u SAT rešavače je poseban problem i više o tome se može pronaći u radovima [13], [14] i [15]. `Python z3 solver` ima funkciju koja omogućava da prosto definišemo ovaj uslov, bez zalaženja u to kako će isti biti predstavljen u KNF formi.

Sledeći uslov koji možemo da uvedemo je jedan od najjednostavnijih. To je uslov koji govori o tome da ako je jedno polje popunjeno, tada polja koja su povezana sa njim po dijagonali sigurno nisu popunjena. Ovaj uslov možemo da zapišemo kao:

$$p_{i,j} \Rightarrow (\neg p_{i+1,j+1} \wedge \neg p_{i-1,j+1} \wedge \neg p_{i+1,j-1} \wedge \neg p_{i-1,j-1}) \tag{4.9}$$

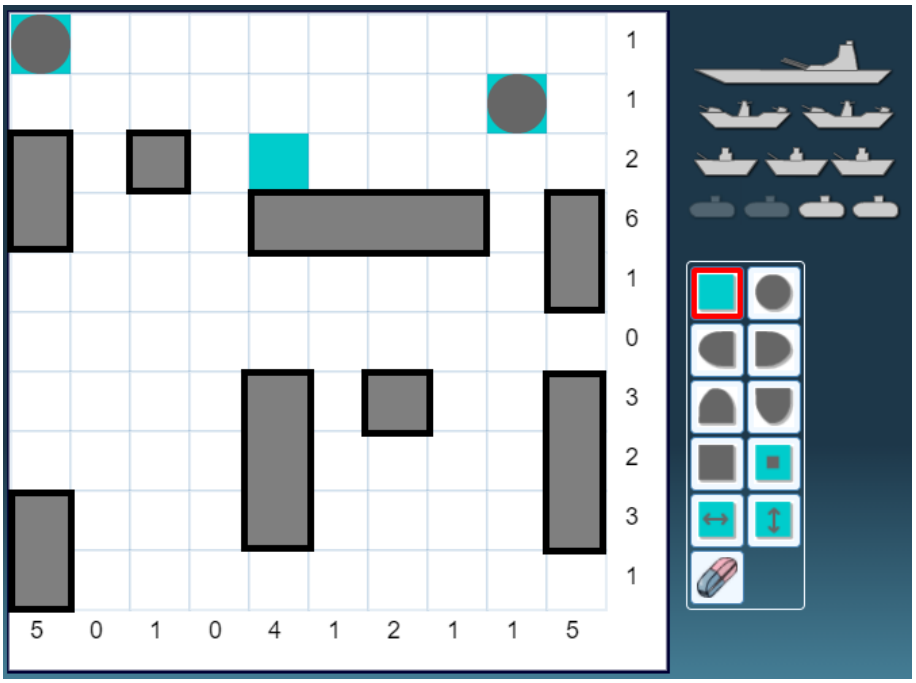
U ovoj formuli i u formulama u nastavku ćemo podrazumevati da indeksi koji se pojavljuju uzimaju samo smislene vrednosti, odnosno vrednosti iz skupa $\{0, 1, \ldots, 9\}$.

Dalje ćemo dodati uslov da nijedna podmornica ne zauzima više od četiri polja:

$$(p_{i,j} \wedge p_{i+1,j} \wedge p_{i+2,j} \wedge p_{i+3,j}) \Rightarrow (\neg p_{i-1,j} \wedge \neg p_{i+4,j}) \tag{4.10}$$

$$(p_{i,j} \wedge p_{i,j+1} \wedge p_{i,j+2} \wedge p_{i,j+3}) \Rightarrow (\neg p_{i,j-1} \wedge \neg p_{i,j+4}) \tag{4.11}$$

Nakon ovoga dodajemo uslov koji govori o tome da ako su dva susedna polja zauzeta, polja koja su sa strane treba da budu slobodna. Kako smo već definisali uslov da polja koja su povezana dijagonalno treba da budu prazna, dovoljno je zahtevati da su polja levo i desno od polja ispred kog je popunjeno polje prazna.



Slika 4.2: Igra podmornica - lakša verzija rešenje

$$\begin{aligned}(p_{i,j} \wedge p_{i+1,j}) &\Rightarrow (\neg p_{i,j-1} \wedge \neg p_{i,j+1}) \\ (p_{i,j} \wedge p_{i,j+1}) &\Rightarrow (\neg p_{i-1,j} \wedge \neg p_{i-1,j})\end{aligned} \tag{4.12}$$

Konačno, dovoljno je dodati uslove koji definišu početno stanje na tabli.

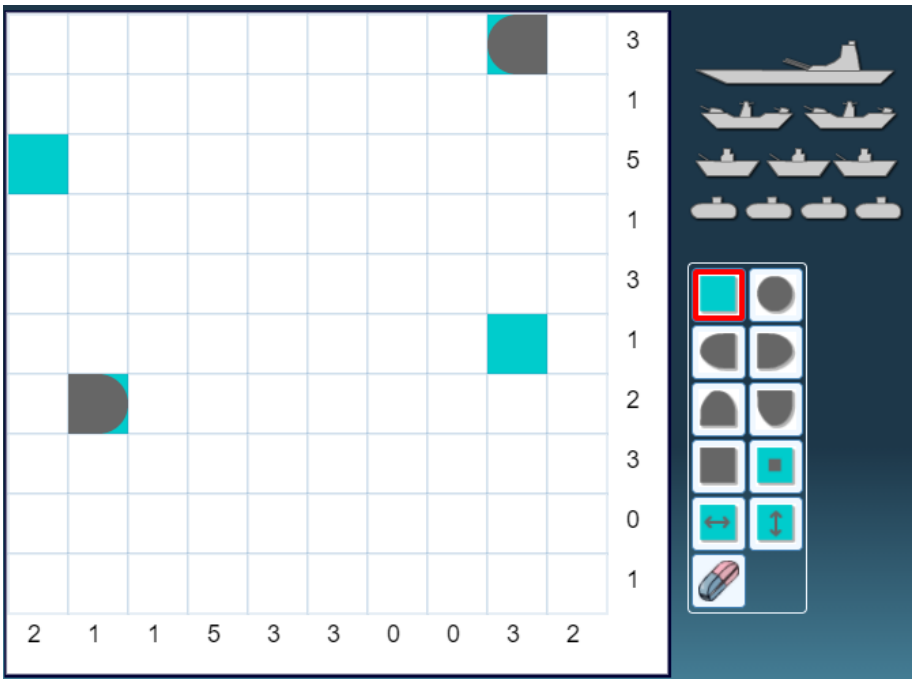
$$p_{0,0} = 1, \quad p_{1,8} = 1, \quad p_{8,0} = 1, \quad p_{9,0} = 1 \tag{4.13}$$

Na slici 4.2 je prikazana tabla koja je popunjena na osnovu dobijenog rešenja. U prilogu B se može videti program i rezultat programa.

4.4 Podmornice - teža verzija igre

Na slici 4.3 je prikazana teža verzija ove igre. U težoj verziji igre je neophodno koristiti informaciju o broju brodova svake vrste da bi se došlo do rešenja.

Da bismo mogli da rešimo težu verziju igre potrebno je da uvedemo dodatne promenljive koje će predstavljati brodove. Podmornice koje zauzimaju jedno polje će biti definisane svojim koordinatama, a njihove promenljive ćemo označiti kao $o_{i,j}$ gde su i i j koordinate. Podmornice koje zauzimaju više od jednog polja ćemo definisati svojm početnim koordinatama i pravcem (vertikalno i horizontalno). Za svaku podmornicu uvodimo promenljive $v_{i,j,k}$ i $h_{i,j,k}$ koje označavaju vertikalni i horizontalni pravac, respektivno, na koordinatama (i, j) veličine k. Odnosno, ako je $h_{0,0,3} = 1$ to će značiti da je podmornica koja je horizontalno usmerena, dužine 3 na polju $(0, 0)$, to jest, polja $(0, 0)$, $(0, 1)$ i $(0, 2)$ su popunjena. Slično, ako je $v_{2,3,2} = 1$, polja $(2, 3)$ i $(2, 4)$ su zauzeta.



Slika 4.3: Igra podmornica - teža verzija

Osim ovih promenljivih imamo i promenljive koje označavaju da je polje popunjeno, u oznaci $p_{i,j}$. Ove promenljive u suštini nisu neophodne da bismo opisali problem, to jest, moguće je problem rešiti i samo korišćenjem promenljivih koje predstavljaju podmornice, ali u tom rešenje postaje nerazumljivo i manje pregledno. Pošto dodatne promeljive ne predstavljaju problem SAT rešavaču, biramo pristup koji obezbeđuje jednostavniji i čitljiviji model.

Za ovako definisane promenljive imamo sledeće uslove:

$$o_{m,n} \Leftrightarrow p_{m,n} \wedge \left(\bigwedge_{i=m-1}^{m+1} \bigwedge_{j=n-1, j\neq i}^{n+1} \neg p_{i,j} \right) \tag{4.14}$$

$$h_{m,n,k} \Leftrightarrow \left(\bigwedge_{j=n}^{n+k-1} p_{m,j} \right) \wedge \neg p_{m,n-1} \wedge \neg p_{m,n+k} \wedge \left(\bigwedge_{j=n-1}^{n+k} \neg p_{m-1,j} \wedge \neg p_{m+1,j} \right) \tag{4.15}$$

$$v_{m,n,k} \Leftrightarrow \left(\bigwedge_{i=m}^{m+k-1} p_{i,n} \right) \wedge \neg p_{m-1,n} \wedge \neg p_{m+k,n} \wedge \left(\bigwedge_{i=m-1}^{m+k} \neg p_{i,n-1} \wedge \neg p_{i,n+1} \right) \tag{4.16}$$

Pošto dati uslovi ostavljaju mogućnost kreiranja figura koje ne izgledaju kao podmornice, dodajemo i uslov o tome da dijagonale oko popunjenog polja moraju biti prazne:

$$p_{i,j} \Rightarrow (\neg p_{i+1,j+1} \wedge \neg p_{i-1,j+1} \wedge \neg p_{i+1,j-1} \wedge \neg p_{i-1,j-1}) \tag{4.17}$$

Da bismo obezbedili da brodovi ne udaraju u zidove dodaćemo uslove:

$$\begin{aligned} h_{i,j,k} &= 0 \quad \forall\, (j + k > 10) \\ v_{i,j,k} &= 0 \quad \forall\, (i + k > 10) \end{aligned} \tag{4.18}$$

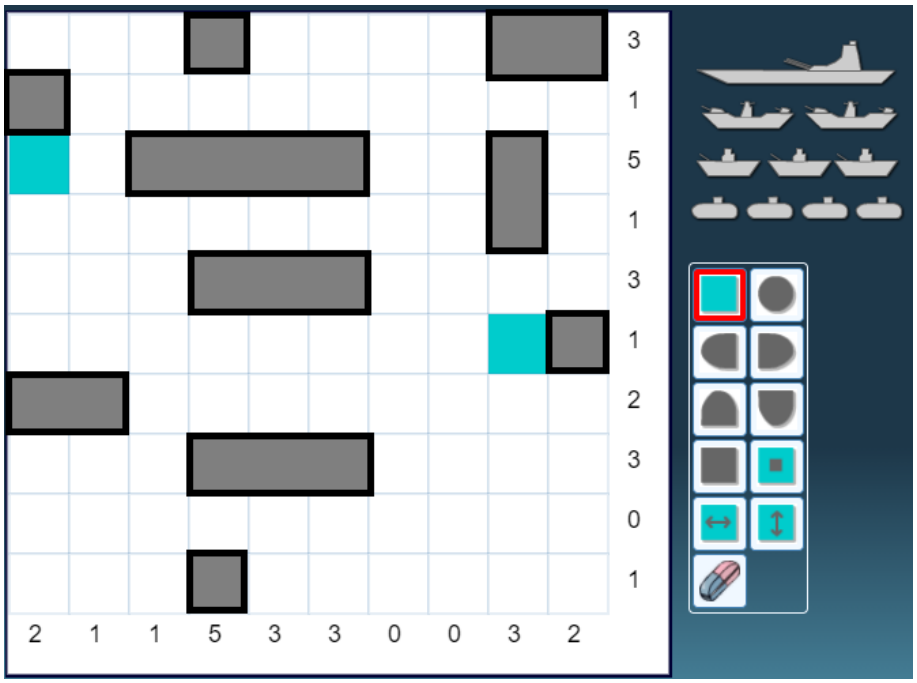
Ponovo dodajemo uslov o broju popunjenih redova i kolona:

$$\sum_{i=0}^{9} p_{i,j} = c_j, \quad \sum_{j=0}^{9} p_{i,j} = r_i \tag{4.19}$$

Na kraju dodajemo uslov koji govori o broju podmornica odgovarajućih veličina:

$$\begin{gathered} \sum_{i=0}^{10} \sum_{j=0}^{10} h_{i,j,k} + v_{i,j,k} = 5 - k, \quad k = 2, 3, 4 \\ \sum_{i=0}^{10} \sum_{j=0}^{10} o_{i,j} = 4 \end{gathered} \tag{4.20}$$

Na osnovu ovako definisanih uslova dobijamo rešenje prikazano na slici 4.4. Programski kôd i rezultat programa su dati u prilogu C.



Slika 4.4: Igra podmornica - teža verzija rešenje

Glava 5

Zaključak

U ovom seminarskom radu je razmatran SAT problem i rešavanje problema svođenjem na SAT problem.

U odeljku 4.2 je ilustrovano svođenje problema na problem SAT tako što je problem direktno formulisan u KNF formi i potom prosleđen SAT rešavaču. Prikazano rešenje se može iskoristiti i za *mathdoku* većih dimenzija, ili se uz manje modifikacije može primeniti da razne vrste igre sudoku. Na ovom problemu je ilustrovano retko kodiranje, biranje odgovarajućih promenljivih i svođenje formula na KNF oblik.

U odeljku 4.3 je prikazano korišćenje `z3py` rešavača i razmatrani su neki kompleksniji uslovi koje je često potrebno predstaviti na odgovarajući način kada pokušavamo problem da svedemo na SAT.

U odeljku 4.4 je prikazano rešenje koje ilustruje predstavljanje uslova koji zahtevaju da određeni broj promenljivih ima neku vrednost. Videli smo kako se uz pravilan izbor promenljivih problem može predstaviti na jednostavan način i kako se takva složenija ograničenja mogu predstaviti. Videli smo, takođe, da je često poželjno uvesti redudantne promenljive, ukoiko to dovodi do jednostavnije predstave problema.

Kroz ove primere smo videli da je prilikom rešavanja problema svođenjem na SAT najbolje uočiti što je više moguće pravilnosti, i da promenljive treba birati tako da se pojave novi šabloni koje možemo da iskoristimo. Konačno, neophodno je precizno i jasno definisati uslove koji karakterišu problem koji rešavamo.

Literatura

[1] Stephen A Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, 1971.

[2] Mladen Nikolić Predrag Janičić. *Veštačka inteligencija*. Matematički fakultet, Beograd, 2021.

[3] Predrag Janičić. *Matematička logika u računarstvu*. Matematički fakultet, Beograd, 2008.

[4] Carla P. Gomes, Henry Kautz, Ashish Sabharwal, and Bart Selman. Satisfiability solvers, 2007.

[5] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535, 2001.

[6] Richard M Stallman and Gerald J Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial intelligence*, 9(2):135–196, 1977.

[7] Niklas Sörensson and Niklas Eén. A sat solver with conflict-clause minimization. *Proceedings of the Theory and Applications of Satisfiability Testing*, 2005.

[8] Carla P Gomes, Bart Selman, Henry Kautz, et al. Boosting combinatorial search through randomization. *AAAI/IAAI*, 98:431–437, 1998.

[9] Z3 api in python. `https://ericpony.github.io/z3py-tutorial/guide-examples.htm`.

[10] Mirko S. Stojanović. *Rešavanje problema CSP tehnikama svođenja na problem SAT*. PhD thesis, Univerzitet u Beogradu, Matematički fakultet, 2016.

[11] Milan Šešum. Svođenje kriptografskih problema na problem sat. Master's thesis, Univerzitet u Beogradu, Matematički fakultet, 2008.

[12] Igor Rodić. Igre parnosti i njihovo rešavanje svođenjem na sat. Master's thesis, Univerzitet u Beogradu, Matematički fakultet, 2020.

[13] I. Abío, R. Nieuwenhuis, A. Oliveras, E. Rodríguez-Carbonell, and V. Mayer-Eichberger. A new look at bdds for pseudo-boolean constraints. *Journal of Artificial Intelligence Research*, 45:443–480, Nov 2012.

[14] A. Frisch and Paul A. Giannaros. Sat encodings of the at-most-k constraint some old , some new , some fast , some slow. 2010.

[15] Neng-Fa Zhou. Yet another comparison of sat encodings for the at-most-k constraint, 2020.

Prilog A : Mathdoku

[12]:
```
f = open("mathdoku.txt", "a")

p = [[i + 3*v + 1 for i in range(3)] for v in range(9)]
# Uzima bar jednu vrednost iz domena.
for v in range(9):
    print(p[v][0], p[v][1], p[v][2], 0, file=f)

# Ne moze imati dve vrednosti istovremeno.
for v in range(9):
    for i in range(3):
        for j in range(i + 1,3):
            if i != j:
                print(-p[v][i], -p[v][j], 0, file=f)

# Ne sme se ponavljati u istom redu i istoj koloni.
for v in range(9):
    for w in range(9):
        if w != v and (w % 3 == v % 3 or w // 3 == v // 3):
            print(-p[v][0], p[w][1], p[w][2], 0, file=f)
            print(-p[v][1], p[w][0], p[w][2], 0, file=f)
            print(-p[v][2], p[w][0], p[w][1], 0, file=f)

# Uslov 6x
B1 = [0, 1, 4]
for v in B1:
    for w in B1:
        if w != v:
            print(-p[v][0], p[w][1], p[w][2], 0, file=f)
            print(-p[v][1], p[w][0], p[w][2], 0, file=f)
            print(-p[v][2], p[w][0], p[w][1], 0, file=f)

# Uslov /2
print(p[3][0], p[3][1], 0, file=f)
print(p[3][1], p[6][1], 0, file=f)
print(p[3][0], p[6][0], 0, file=f)
print(p[6][0], p[6][1], 0, file=f)

# Uslov /3
print(p[2][0], p[2][2], 0, file=f)
```

```
print(p[2][2], p[5][2], 0, file=f)
print(p[2][0], p[5][0], 0, file=f)
print(p[5][0], p[5][2], 0, file=f)

# Uslov -1
print(-p[7][1], p[8][0], p[8][2], 0, file=f)
print(-p[8][1], p[7][0], p[7][2], 0, file=f)
print(p[7][1], p[8][1], 0, file=f)

f.close()
```

[10]:
```
# Resenje dobijeno koriscenjem SAT resavaca https://msoos.github.io/
 ↪cryptominisat_web/
sol = [-1, -2, 3,       # p0 = 3
       -4, 5, -6,       # p1 = 2
       7, -8, -9,       # p2 = 1
       -10, 11, -12,    # p3 = 2
       13, -14, -15,    # p4 = 1
       -16, -17, 18,    # p5 = 3
       19, -20, -21,    # p6 = 1
       -22, -23, 24,    # p7 = 3
       -25, 26, -27]    # p8 = 2
```

[ ]:

Prilog B : Yubotu (lakša verzija)

[1]:
```
import z3
from z3 import Implies, Not, And, Bool, PbEq, sat
```

[2]:
```
import matplotlib.pyplot as plt

def plot_model(m):
    mat = [[(i-j)%2 for i in range(10)] for j in range(10)]
    for i in range(10):
        for j in range(10):
            if m.evaluate(p[i][j]):
                mat[i][j] = 10 + (i-j) % 2
    plt.matshow(mat)
```

[3]:
```
s = z3.Solver()
p = [[z3.Bool("p{},{}".format(j, i)) for i in range(10)] for j in range(10)]

rows = [1, 1, 2, 6, 1, 0, 3, 2, 3, 1]
col = [5, 0, 1, 0, 4, 1, 2, 1, 1, 5]
```

[4]:
```
# Broj popunjenih polja u redovima.
for (i, r) in enumerate(rows):
    s.add(z3.PbEq([(p[i][j],1) for j in range(10)], r))

# Broj popunjenih polja u kolonama.
for (j, c) in enumerate(col):
    s.add(z3.PbEq([(p[i][j],1) for i in range(10)], c))
```

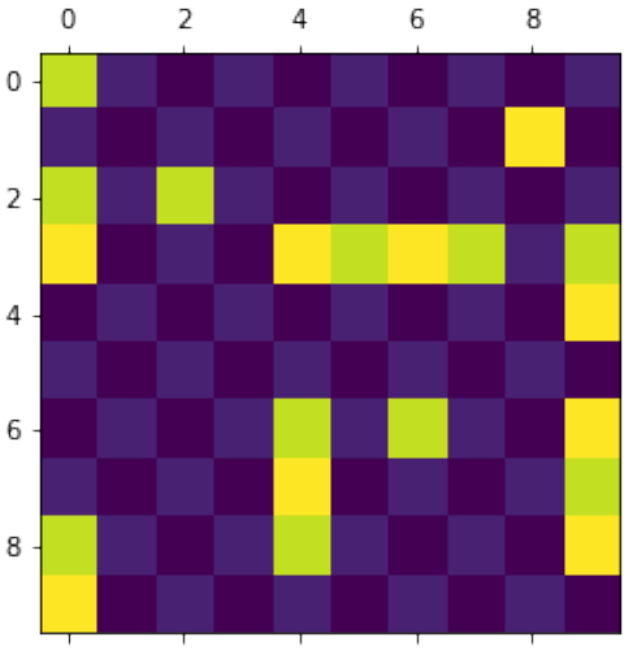
[5]:
```
# Dijagonalni uslov.
for i in range(10):
    for j in range(10):
        if (i < 9 and j < 9):
            s.add(Implies(p[i][j], Not(p[i+1][j+1])))
        if (i < 9 and j > 0):
            s.add(Implies(p[i][j], Not(p[i+1][j-1])))
        if (i > 0 and j < 9):
            s.add(Implies(p[i][j], Not(p[i-1][j+1])))
        if (i > 0 and j > 0):
            s.add(Implies(p[i][j], Not(p[i-1][j-1])))
```

[6]:
```
# Maksimalno 4 polja uslov - redovi.
for i in range(6):
    for j in range(10):
        s.add(Implies(And(p[i][j], p[i+1][j], p[i+2][j], p[i+3][j]),␣
 ↪Not(p[i+4][j])))
        if i > 0:
            s.add(Implies(And(p[i][j], p[i+1][j], p[i+2][j], p[i+3][j]),␣
 ↪Not(p[i-1][j])))

# Maksimalno 4 polja uslov - kolone.
for i in range(10):
    for j in range(6):
        s.add(Implies(And(p[i][j], p[i][j+1], p[i][j+2], p[i][j+3]),␣
 ↪Not(p[i][j+4])))
        if j > 0:
            s.add(Implies(And(p[i][j], p[i][j+1], p[i][j+2], p[i][j+3]),␣
 ↪Not(p[i][j-1])))
```

[7]:
```
# Polja desno i levo prazna ako je polje ispred zauzeto - redovi.
for i in range(9):
    for j in range(10):
        if j > 0:
            s.add(Implies(And(p[i][j], p[i+1][j]), Not(p[i][j-1])))
        if j < 9:
            s.add(Implies(And(p[i][j], p[i+1][j]), Not(p[i][j+1])))

# # Polja desno i levo prazna ako je polje ispred zauzeto - redovi.
for i in range(10):
    for j in range(9):
        if i > 0:
            s.add(Implies(And(p[i][j], p[i][j+1]), Not(p[i-1][j])))
        if i < 9:
            s.add(Implies(And(p[i][j], p[i][j+1]), Not(p[i+1][j])))
```

[8]:
```
# Pocetno stanje igre.
s.add(p[0][0])
s.add(p[1][8])
s.add(p[8][0])
s.add(p[9][0])
```

[9]:
```
s.set(unsat_core=True)
if s.check() == sat:
    m = s.model()
    plot_model(m)
else:
    print(s.unsat_core())
```


0
2
4
6
8
0
2
4
6
8

[ ]:

Prilog C : Yubotu (teža verzija)

[1]:
```
import z3
from z3 import Implies, Not, And, Bool, PbEq, sat
```

[2]:
```
import matplotlib.pyplot as plt

def plot_model(m):
    mat = [[(i-j)%2 for i in range(10)] for j in range(10)]
    for i in range(10):
        for j in range(10):
            if m.evaluate(p[i][j]):
                mat[i][j] = 10 + (i-j) % 2
    plt.matshow(mat)
```

[3]:
```
s = z3.Solver()
p = [[Bool("p{},{}".format(i, j)) for j in range(10)] for i in range(10)]
o = [[Bool("o{},{}".format(i, j)) for j in range(10)] for i in range(10)]
v = [[[Bool("v{},{},{}".format(i, j, k)) for k in range(2, 5)] for j in␣
 ↪range(10)] for i in range(10)]
h = [[[Bool("h{},{},{}".format(i, j, k)) for k in range(2, 5)] for j in␣
 ↪range(10)] for i in range(10)]


rows = [3, 1, 5, 1, 3, 1, 2, 3, 0, 1]
col = [2, 1, 1, 5, 3, 3, 0, 0, 3, 2]
```

[4]:
```
# Broj popunjenih polja u redovima.
for (i, r) in enumerate(rows):
    s.add(z3.PbEq([(p[i][j],1) for j in range(10)], r))

# Broj popunjenih polja u kolonama.
for (j, c) in enumerate(col):
    s.add(z3.PbEq([(p[i][j],1) for i in range(10)], c))
```

[5]:
```
# Dijagonalni uslov.
for i in range(10):
    for j in range(10):
        if (i < 9 and j < 9):
            s.add(Implies(p[i][j], Not(p[i+1][j+1])))
        if (i < 9 and j > 0):
```

```
            s.add(Implies(p[i][j], Not(p[i+1][j-1])))
        if (i > 0 and j < 9):
            s.add(Implies(p[i][j], Not(p[i-1][j+1])))
        if (i > 0 and j > 0):
            s.add(Implies(p[i][j], Not(p[i-1][j-1])))
```

[6]:
```
# Definisanje najmanjih podmornica.
for m in range(10):
    for n in range(10):
        vec = [p[m][n]]
        for i in range(m-1, m+1+1):
            for j in range(n-1, n+1+1):
                if (0 <= i <= 9) and (0 <= j <= 9) and ((i, j) != (m, n)):
                    vec += [Not(p[i][j])]

        s.assert_and_track(Implies(o[m][n], And(vec)), str(o[m][n]) + " -> ")
        s.assert_and_track(Implies(And(vec), o[m][n]), "-> " + str(o[m][n]))
```

[7]:
```
# Definisanje horizontalnih podmornica.
for m in range(10):
    for n in range(10):
        for k in range(2, 5):

            # Uslov da ne udaraju u zid.
            if n + k > 10:
                s.add(Not(h[m][n][k-2]))
                continue

            vec = []

            if 0 <= n-1 <= 9:
                vec += [Not(p[m][n-1])]
            if 0 <= n+k <= 9:
                vec += [Not(p[m][n+k])]

            vec += [p[m][j] for j in range(n, n+k)]

            if 0 <= m-1 <= 9:
                vec += [Not(p[m-1][j]) for j in range(n-1, n+k+1) if 0 <= j <= 9]
            if 0 <= m+1 <= 9:
                vec += [Not(p[m+1][j]) for j in range(n-1, n+k+1) if 0 <= j <= 9]

            s.assert_and_track(Implies(h[m][n][k-2], And(vec)),␣
↪str(h[m][n][k-2]) + '->')
            s.assert_and_track(Implies(And(vec), h[m][n][k-2]), '->' +␣
↪str(h[m][n][k-2]))
```

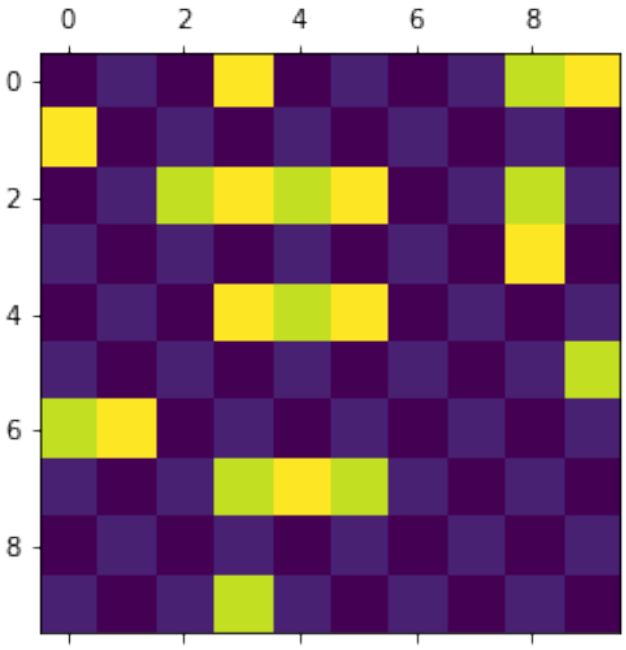
[8]:
```
# Definisanje vertikalnih podmornica.
for m in range(10):
    for n in range(10):
        for k in range(2, 5):

            # Uslov da ne udaraju u zid.
            if m + k > 10:
                s.add(Not(v[m][n][k-2]))
                continue

            vec = []

            if 0 <= m-1 <= 9:
                vec += [Not(p[m-1][n])]
            if 0 <= m+k <= 9:
                vec += [Not(p[m+k][n])]

            vec += [p[i][n] for i in range(m, m+k)]

            if 0 <= n-1 <= 9:
                vec += [Not(p[i][n-1]) for i in range(m-1, m+k+1) if 0 <= i <= 9]
            if 0 <= n+1 <= 9:
                vec += [Not(p[i][n+1]) for i in range(m-1, m+k+1) if 0 <= i <= 9]

            s.assert_and_track(Implies(v[m][n][k-2], And(vec)),␣
↪str(v[m][n][k-2]) + '->')
            s.assert_and_track(Implies(And(vec), v[m][n][k-2]), '->' +␣
↪str(v[m][n][k-2]))
```

[9]:
```
# Broj odgovarajucih podmornica.
for k in range(2, 5):
    vec = []
    for i in range(10):
        for j in range(10):
            vec += [(h[i][j][k-2], 1)]
            vec += [(v[i][j][k-2], 1)]
    s.assert_and_track(PbEq(vec, 5-k), 'k='+str(k))

vec = []
for i in range(10):
    vec += [(o[i][j], 1) for j in range(10)]
s.assert_and_track(PbEq(vec, 4), 'k=1')
```

[10]:
```
# Pocetni uslovi igre.
s.add(h[6][0][2-2])
s.add(h[0][8][2-2])
```

```
s.add(Not(p[2][0]))
s.add(Not(p[5][8]))
```

[11]:
```
s.set(unsat_core=True)
if s.check() == sat:
    m = s.model()
    plot_model(m)
else:
    print(s.unsat_core())
```



[ ]: