Binarni dijagrami odlučivanja, implementacija

Marija Bogavac 1068/23

7. mart 2025.

Sažetak

Ovaj rad se bavi implementacijom binarnog dijagrama odlučivanja (BDD) kao metode za rešavanje SAT problema. Binarni dijagrami odlučivanja pružaju efikasan način za predstavljanje logičkih izraza u vidu grafa, često smanjujući eksponencijalnu složenost njihove evaluacije. Takođe, diskutuje se o primenama BDD-a u različitim oblastima, poređenju sa DPLL-om i, naravno, implementacija BDD-a u jeziku Python sa dodatnim funkcijama koje rešavaju da li je formula zadovoljiva i koja je valuacija zadovoljava.

1 Uvod

SAT problem je problem ispitivanja da li za datu iskaznu formulu postoji dodela vrednosti iskaznim atomima takva da formula bude tačna.

Na primer: $p_1 \wedge p_2$ je zadovoljiva formula, jer postoji valuacija $(p_1 = T, p_2 = T)$ za koju je tačna.

SAT je NP-kompletan problem, pa su svi postojeći algoritmi za njegovo rešavanje eksponencijalne vremenske složenosti u najgorem slučaju prilikom pronalaženja odgovarajuće dodele (valuacije) atoma koja zadovoljava formulu – težak problem. Provera da li je neka predložena valuacija tačna može se izvršiti u polinomijalnom vremenu – lak problem.

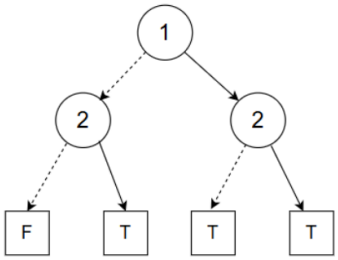
Provera zadovoljivosti formule može se efikasno realizovati korišćenjem binarnog dijagrama odlučivanja u mnogim praktičnim slučajevima, iako proces konstrukcije može imati eksponencijalnu složenost.

2 Osnove

BDD je reprezentacija logičkog izraza u obliku usmerenog acikličnog grafa sa korenom. Graf sadrži dve vrste čvorova:

- listove koji predstavljaju vrednosti tačno ili netačno,
- odlučujući čvorovi koji sadrže jedan atom formule i imaju dva potomka čvora – niži i viši.

Niži označava da je njegovom roditelju dodeljena vrednost 0 (na grafu se grana ka njemu crta isprekidanom linijom), a viši označava da je njegovom roditelju dodeljena vrednost 1.

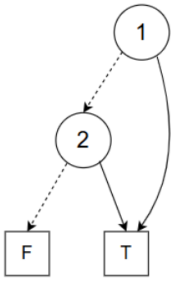


Slika 1: BDD za $p_1 \vee p_2$

BDD je uredjen ako se različiti atomi pojavljuju u istom redosledu na svim putanjama od korena tj. na svim putanjama se poštuje linearni redosled $p_1 < p_2 < \ldots < p_n$ BDD je redukovan ako su ispunjena dva uslova:

1. Ne postoje dva različita čvora u i v koji imaju isto ime promenljive (atoma) u sebi, nižeg (low) i višeg ($high$) potomka : $var(u) = var(v), low(u) = low(v), high(u) = high(v)$ implicira $u = v$
2. Ne postoji čvor koji ima identičnog nižeg i višeg potomka $low(u) \neq high(u)$

ROBDD je redukovan uredjen BDD i on je jedinstven za svaku formulu do na redosled promenljivih. U nastavku ćemo razmatrati isključivo ROBDD, iako ćemo, zbog jednostavnosti koristiti oznaku BDD.
Takođe, nema potrebe za više od 2 lista (jedan True i jedan False list).



Slika 2: ROBDD za $p_1 \vee p_2$

2.1 Problem redosleda promenljivih

Ako redosled promenljivih nije optimalan, broj čvorova u grafu može eksponencijalno porasti. Problem nalaženja dobrog redosleda promenljivih je NP-težak, ali postoje određene heuristike koje mogu približno dobro rešiti problem:

- **Statički poredak** – traženje dobrog redosleda promenljivih pre konstrukcije BDD-a.
  Praksa je da pri vrhu budu promenljive manjeg indeksa, a da se indeksi povećavaju ka listovima drveta. Ovo ne mora uvek da dovede do najmanjeg drveta, ali se generalno koristi (korišćeno je u implementaciji).
  Postoje algoritmi koji analiziraju zavisnosti tj. veze između promenljivih u formuli (topološka analiza). Od formule se prvo pravi graf povezanosti modela (engl. model connectivity graph) čije su grane zavisnosti između promenljivih, a čvorovi promenljive. Grane mogu da imaju i svoju težinu u odnosu na jačinu veze pa se mogu koristiti algoritmi topološkog sortiranja ili klasterovanja (promenljive sa jakim vezama se grupišu) da bi se utvrdio što bolji redosled.

- **Dinamički poredak** – preuređivanje promenljivih u potrazi za manjim BDD-om kada graf nadmaši neku unapred određenu veličinu. Jedan algoritam koji dinamički preuređuje BDD zove se 'Sifting'. Ideja algoritma je isprobati poziciju za svaku promenljivu i postaviti je u grafu tako da minimizuje veličinu BDD-a.

3 Opis metode

Svaka logička formula može se izraziti preko If-then-else operatora (ITE) i konstanti 0 i 1 koje predstavljaju netačno i tačno.

If-then-else operator je definisan kao

$$ITE(p_1, p_2, p_3) = (p_1 \wedge p_2) \vee (\neg p_1 \wedge p_3) \tag{1}$$

odnosno, ako je p_1 tačno onda $ITE(p_1, p_2, p_3) = p_2$ jer je izraz $(p_1 \wedge p_2)$ tačan; ako je p_1 netačno onda $ITE(p_1, p_2, p_3) = p_3$ jer je izraz $(\neg p_1 \wedge p_3)$ tačan.

Operacija	ITE operator
$f \wedge g$	$ITE(f, g, 0)$
$f \vee g$	$ITE(f, 1, g)$
$f \rightarrow g$	$ITE(f, g, 1)$
$f \leftrightarrow g$	$ITE(f, g, \neg g)$
$f \oplus g$	$ITE(f, \neg g, g)$
$\neg f$	$ITE(f, 0, 1)$

Tabela 1: Logicke operacije predstavljene preko ITE operatora

Čvor grafa je predstavljen kao klasa sa poljima:

- i - indeks (redni broj) atoma (promenljive $p_0 < p_1 < \ldots < p_{127}$). Tip podataka je int.
- high - predstavlja čvor u koji se odlazi ako je trenutni čvor evaluiran na 1 (na slikama 1 i 2 puna linija). Tip podataka je Optional['BDD'] što znači da može sadržati None ili čvor.
- low - predstavlja čvor u koji se odlazi ako je trenutni čvor evaluiran na 0 (na slikama 1 i 2 isprekidana linija). Tip podataka je takođe Optional['BDD'].

Čvor drveta takođe predstavlja i samo drvo (jer je moguć obilazak od bilo kog čvora do listova).

- U *bdd.py* nalaze se metodi za pravljenje BDD-a. Čvorovi se skladište u listi. Bitno je da nema dupliranih čvorova (zbog poštovanja pravila ROBDD-a) tako da će se to proveravati u funkciji create_node.

Logičke operacije bdd_and, bdd_or, bdd_xor, bdd_not, bdd_imp, bdd_eq su napisane po ugledu na tabelu 1.

Ispis je po nivoima, korišćen je BFS (izgleda jednostavno i ima ponavljanja listova T,F i drugih promenljvih) u metodi print_bdd.

Metod compute_value postavlja vrednost prosleđene promenljive na tačno ili netačno i tako dalje redukuje drvo i propagira promenu dublje do listova praveći nove čvorove.

Glavni metod je ITE koji koristi rekurziju i biće detaljnije objašnjen.

- U *sat_solve.py* nalaze se metodi za rešavanje problema SAT.

Metod SAT_solver ispisuje da li je prosleđena logička formula zadovoljiva ili nije. Koristi se rekurzija u potrazi za tačnim listovima.

Metod SAT_valuations ispisuje dodelu vrednosti True/False promenljivama za koje će konačna formula biti tačna ili vraća None. Takođe je rekurzivna i vraća rezultat ako se stigne do tačnog lista.

4 Implementacija

4.1 bdd.py

Klasa za BDD čvor:

```
class BDD:
    def __init__(self, i: int, high: Optional['BDD'], low:
        Optional['BDD']):
        self.i = i
        self.high = high
        self.low = low
```

Globalna promenljiva koja služi za skladištenje čvorova je listOfNodes. Inicijalizovana je tako da sadrži false čvor BDD(129, None, None) i true čvor BDD(130, None, None).
Brojevi 129 i 130 su izabrani tako da ne dolazi do konflikta sa indeksima promenljivih koji idu od 0 do 127, što znači da je moguće uneti formulu koja sadrži 128 promenljivih.

```
listOfNodes: List = [BDD(129, None, None), BDD(130, None, None)]
```

Radi bolje preglednosti biće korišćene funkcije bdd_false() i bdd_true() za označavanje tačnog i netačnog lista.
Lista listOfNodes biće korišćena u create_node funkciji.

```
def bdd_false() -> BDD:
    return listOfNodes[0]

def bdd_true() -> BDD:
    return listOfNodes[1]
```

U main.py postoji inicijalizacija atoma koji će biti korišćeni u logičkim izrazima:

```
    p1 = add_new_variable(1)
    p2 = add_new_variable(2)
    p3 = add_new_variable(3)
```

Ti atomi se procesiraju i od njih se prave čvorovi. Inicijalno svaki čvor će pokazivati high granom na tačni list, a low granom na netačni list.
Ako je atom već unet (recimo pojavljuje se dvaput u formuli) onda je potrebno vratiti taj isti čvor (sprečavamo duplikate). Inače se pravi novi čvor.
Metod create_node će se koristiti na još jednom mestu.

```
def create_node(index: int, high: Optional[BDD], low: Optional[BDD]) ->
    BDD:
    for node in listOfNodes:
        if node.i == index and node.high == high and node.low == low:
            return node
    new_node = BDD(index, high, low)
    listOfNodes.append(new_node)
    return new_node

def add_new_variable(i: int) -> BDD:
    return create_node(i, bdd_true(), bdd_false())
```

Metod compute_value – redukuje BDD tako sto nekom atomu dodeljuje vrednost true/false.
Pošto je pravilo statičkog poretka da manji indeksi budu pri vrhu drveta onda je prvi if jasan.

Ako je indeks trenutnog čvora manji od indeksa čvora kog želimo da evaluiramo onda rekurzivno prolazimo kroz decu trenutnog čvora i pravimo novi čvor od trenutnog čija će deca biti redukovana (jer je evaluiran jedan atom).
Ako je indeks trenutnog čvora jednak indeksu čvora kog želimo da evaluiramo onda se dalje kroz drvo krećemo preko high grane (ako je evaluacija čvora tačna) ili preko low grane (ako je evaluacija čvora netačna).

```
def compute_value(subtree: Optional[BDD], index: int, value: bool) ->
    Optional[BDD]:
    if subtree is None:
        return None
    if subtree.i > index: # Index se ne pojavljuje u poddrvetu
        return subtree
    elif subtree.i < index:
        return create_node(
            subtree.i,
            compute_value(subtree.high, index, value),
            compute_value(subtree.low, index, value),
        )
    else: # subtree.i == index
        return compute_value(subtree.high if value else subtree.low,
            index, value)
```

Metod ITE - ako je if_node evaluiran na true onda ćemo nastaviti obilazak preko then_node;
ako je if_node evaluiran na false onda ćemo nastaviti obilazak preko else_node.
Bazni slučajevi su najjednostavniji slučajevi.
Biramo smallest_index jer nam je bitan poredak promenljivih, da manji indeksi budu pri vrhu.
Cilj je da rekurzivno napravimo BDD u odnosu na to da li je izabrani čvor true ili false.
Gradimo dva poddrveta – jedno ako se smallest_index evaluira na true, drugo ako se evaluira na false.
Na kraju čvor sa smallest_index pokazuje na ta dva poddrveta (high je za evaluaciju na true, low je za evaluaciju na false).

```
def ITE(if_node: Optional[BDD], then_node: Optional[BDD], else_node:
    Optional[BDD]) -> Optional[BDD]:

    # Bazni slucajevi
    if if_node == bdd_true():
        return then_node
    if if_node == bdd_false():
        return else_node
    if then_node == else_node:
        return then_node
    if then_node == bdd_true() and else_node == bdd_false():
        return if_node

    # Biramo najmanji (topmost) atom - zbog problema orderinga,
        potrebno je da se indeksi povecavaju iduci ka dnu drveta
    smallest_index = min(if_node.i, then_node.i,else_node.i)

    # Atom sa smallest index je true
    if_node_computed_true = compute_value(if_node, smallest_index, True)
    then_node_computed_true = compute_value(then_node, smallest_index,
        True)
    else_node_computed_true = compute_value(else_node, smallest_index,
        True)
    ITE_true = ITE(if_node_computed_true, then_node_computed_true,
        else_node_computed_true)
```

```
    # Atom sa smallest index je false
    if_node_computed_false = compute_value(if_node, smallest_index,
        False)
    then_node_computed_false = compute_value(then_node, smallest_index,
        False)
    else_node_computed_false = compute_value(else_node, smallest_index,
        False)
    ITE_false = ITE(if_node_computed_false, then_node_computed_false,
        else_node_computed_false)

    return create_node(smallest_index, ITE_true, ITE_false)
```

Logičke operacije:

```
def bdd_not(f: Optional[BDD]) -> Optional[BDD]:
    return ITE(f, bdd_false(), bdd_true())

def bdd_and(f: Optional[BDD], g: Optional[BDD]) -> Optional[BDD]:
    return ITE(f, g, bdd_false())

def bdd_or(f: Optional[BDD], g: Optional[BDD]) -> Optional[BDD]:
    return ITE(f, bdd_true(), g)

def bdd_xor(f: Optional[BDD], g: Optional[BDD]) -> Optional[BDD]:
    return ITE(f, bdd_not(g), g)

def bdd_eq(f: Optional[BDD], g: Optional[BDD]) -> Optional[BDD]:
    return ITE(f, g, bdd_not(g))

def bdd_imp(f: Optional[BDD], g: Optional[BDD]) -> Optional[BDD]:
    return ITE(f, g, bdd_true())
```

Metod print_bdd koristi BFS i ide kroz nivoe, ispisuje na šta pokazuje prethodni nivo. Ima ponavljanja čvorova T,F u ispisu iako u memoriji postoje samo dva lista T i F. Tako da ova funkcija prikazuje samo približno kako drvo izgleda - nije u potpunosti redukovano (ROBDD) u prikazu.

```
def print_bdd(root: Optional[BDD]):
    if root is None:
        return
    queue = [root]
    while queue:
        level_size = len(queue)
        for _ in range(level_size):
            current = queue.pop(0)
            if current == bdd_true():
                print("T ", end="")
            elif current == bdd_false():
                print("F ", end="")
            else:
                print(f"p{current.i} ", end="")
                if current.low:
                    queue.append(current.low)
                if current.high:
                    queue.append(current.high)
        print()
```

4.2 sat_solve.py

SAT_solver – ispituje da li formula predstavljena preko bdd-a zadovoljiva. Ova funkcija vraća true/false.
Princip rada je rekurzivno kretanje kroz BDD ka true listu izbegavajući puteve koji vode u false list.

```
def solver_rec(tree: Optional[BDD]):
    if tree is None:
        return False

    # Bazni slucajevi - dosli smo do listova
    if tree == bdd_true():
        return True
    if tree == bdd_false():
        return False

    # proveri high i low granu, da li neka vodi ka true listu
    high_result = solver_rec(tree.high)
    low_result = solver_rec(tree.low)

    return high_result or low_result

def SAT_solver(tree: Optional[BDD]) -> bool:
    if tree is None or tree == bdd_false():
        return False
    return solver_rec(tree)
```

Metoda find_valuations rekurzivno traži prvu zadovoljavajuću valuaciju promenljivih u BDD. Ako naiđe na čvor koji predstavlja True, vraća trenutnu dodelu vrednosti true/false promenljivama, dok u slučaju False vraća None.
Metoda SAT_valuations poziva find_valuations sa praznom mapom kako bi pronašla bilo koje valuaciju za dati BDD.

```
def find_valuations(tree: Optional[BDD], valuations: Dict[int, bool])
    -> Optional[Dict[int, bool]]:
    if tree is None:
        return None

    # Bazni slucaj: dosli smo do T, vrati trenutnu valuaciju
        promneljivih
    if tree == bdd_true():
        return valuations
    # Bazni slucaj: dosli smo do F, nema valuacije za koje je formula
        zadovoljiva
    if tree == bdd_false():
        return None

    # trenutna promenljiva se evaluira na T
    valuations[tree.i] = True
    result = find_valuations(tree.high, valuations.copy())
    if result is not None: # tree == bdd_true()
        return result

    # trenutna promenljiva se evaluira na F
    valuations[tree.i] = False
    result = find_valuations(tree.low, valuations.copy())
    if result is not None: # tree == bdd_true()
        return result

    return None
```

```
def SAT_valuations(tree: Optional[BDD]) -> Optional[Dict[int, bool]]:
    if tree is None or tree == bdd_false():
        return None
    return find_valuations(tree, {})
```

4.3 main.py

U ovom fajlu nalaze se primeri nekih logičkih izraza, njihovo pretvaranje u BDD i rešavanje SAT.

```
from bdd import add_new_variable, print_bdd, bdd_or, bdd_and, bdd_not,
    bdd_xor, bdd_imp, bdd_eq
from sat_solve import SAT_solver, SAT_valuations

def main():
    p1 = add_new_variable(1)
    p2 = add_new_variable(2)
    p3 = add_new_variable(3)

    tree1 = bdd_and(p1, p2)
    print_bdd(tree1)
    print("Formula p1 & p2 is satisfiable:", SAT_solver(tree1))
    print(SAT_valuations(tree1))
    print("____________________")

    tree2 = bdd_and(p1, bdd_not(p1))
    print_bdd(tree2)
    print("Formula p1 & ~p1 is satisfiable:", SAT_solver(tree2))
    print(SAT_valuations(tree2))
    print("____________________")

    print("BDD for p1 | p2 | p3:")
    tree3 = bdd_or(bdd_or(p1, p2), p3)
    print_bdd(tree3)
    print("Formula p1 | p2 | p3 is satisfiable:", SAT_solver(tree3))
    print(SAT_valuations(tree3))
    print("____________________")

    print("BDD for (p1 | p2) & (‘~p1 | p2 ) & ~p2")
    tree4 =
        bdd_and(bdd_and(bdd_or(p1,p2),bdd_or(bdd_not(p1),p2)),bdd_not(p2))
    print_bdd(tree4)
    print("Formula (p1 | p2) & (‘~p1 | p2 ) & ~p2 is satisfiable:",
        SAT_solver(tree4))
    print(SAT_valuations(tree4))
    print("____________________")

    print("BDD for (p1 | p2) ^ p3:")
    tree5 = bdd_xor(bdd_or(p1, p2), p3)
    print_bdd(tree5)
    print("Formula (p1 | p2) ^ p3 is satisfiable:", SAT_solver(tree5))
    print(SAT_valuations(tree5))
    print("____________________")

    print("BDD for p1 => p2:")
    tree6 = bdd_imp(p1, p2)
    print_bdd(tree6)
    print("Formula (p1 => p2) is satisfiable:", SAT_solver(tree6))
    print(SAT_valuations(tree6))
    print("____________________")
```

```
    # test your input
    '''
    tree =
    print_bdd(tree)
    print("Formula is satisfiable:", SAT_solver(tree))
    print(SAT_valuations(tree))
    '''

if __name__ == "__main__":
    main()
```

5 Zaključak

5.1 Odnos BDD-a sa drugim pristupima rešavanju SAT-a

BDD se često koristi kao alternativa DPLL-u (algoritam za rešavanje SAT problema koji koristi rekurzivnu metodu za razdvajanje i testiranje svih mogućih dodela vrednosti promenljivama) jer pruža bolju kompaktnost u prikazu logičkih funkcija, čime može smanjiti potrebu za brojnim probama dodela i čini proces rešavanja bržim.
Za određene logičke formule BDD je efikasan, ali ako logička funkcija postane vrlo kompleksna ili ima mnogo promenljivih sa malim stepenom simetrije, BDD može postati vrlo veliki i neefikasan. U takvim slučajevima, DPLL i drugi algoritmi mogu biti uspešniji.
Takođe, uz konstrukciju BDD-a za veće logičke izraze uvek postoji težak problem nalaženja dobrog redosleda promenljivih u drvetu.

5.2 Primene binarnih dijagrama odlučivanja

Binarni dijagrami odlučivanja imaju ključnu ulogu u različitim oblastima, uključujući projektovanje digitalnih kola, formalnu verifikaciju, veštačku inteligenciju i kombinatornu optimizaciju. Primeri:

- Provera da li dve logičke funkcije daju identične izlaze (npr. optimizovano kolo i njegova specifikacija).
- Identifikacija logičkih grešaka u dizajnu kola.
- Kompaktno skladištenje stanja sistema, omogućavajući proveru velikih sistema.
- Koristi se u oblasti veštačke inteligencije za predstavljanje znanja i donošenje odluka (pojednostavljivanju Bajesovih mreža i modela odlučivanja)

.

Literatura

[1] Crtanje grafova. https://app.diagrams.net/

[2] YouTube. *Binary Decision Diagrams (BDDs) - Introduction.* https://www.youtube.com/watch?v=m8a3uQEAHl0

[3] YouTube. *Stanford Lecture: Donald Knuth - "Fun With Binary Decision Diagrams (BDDs)"(June 5, 2008)* https://www.youtube.com/watch?v=SQE21efsf7Y

[4] *Implementacija binarnih dijagrama odluka u C++.* https://www.researchgate.net/publication/236149939_Implementing_Binary_Decision_Diagram

[5] Andersen, Henrik Reif. *An Introduction to Binary Decision Diagrams* Dostupno na: https://www.tifr.res.in/~shibashis.guha/courses/diwali2022/andersen-bdd.pdf

[6] Dynamic ordering, Pistek, Martin. *Dynamic variable reordering for Binary Decision Diagrams* https://essay.utwente.nl/96753/1/Pistek_MA_EEMCS.pdf

[7] Static ordering *Learning to Order BDD Variables in Verification* https://arxiv.org/pdf/1107.0020