BDD Dokumentácia

Marek Smutný

Opis riešenia (funkcií):

V mojom riešení som použil dve triedy: Node a Tree. Trieda Node reprezentuje jeden prvok v strome a jej parametre sú string, left a right. Tree má iba parametre root a pocet.

```
class Node:
    def __init__(self, string):
        self.string = string
        self.left = None
        self.right = None


class Tree:
    def __init__(self):
        self.root = None
        self.pocet = 0
```

Na vytváraní stromu sa podieľa veľa funkcií. BDD_create zavolá funkciu „vytvor“ a po vytvorení stromu zavolá funkciu „count_nodes“.

```
#vytvorenie stromu a spocitanie nodes
def BDD_create(tree, postup, maximum, nula, jednotka):
    tree.vytvor(tree.root, 0, maximum, nula, jednotka, postup)
    tree.pocet = len(tree.count_nodes(tree.root, []))
    for i in (tree.count_nodes(tree.root, [])):
        print (i.string)
```

Vo funkcii vytvor funkcia preorder strom vytvára po leveloch a funkcie redukcia_I a redukcia_S ho redukujú.

```
#vytvorenie celeho stromu
def vytvor(self, node, level, maximum, nula, jednotka, postup):
    self.preorder(node, level, 0, nula, jednotka, postup)
    self.redukcia_I(node, level + 1 , 0, [], None)
    if level < maximum:
        self.vytvor(node, level + 1, maximum, nula, jednotka, postup)
    self.redukcia_S(node, None)

#vytvaranie stromu po leveloch
def preorder(self, node, i, j, nula, jednotka, postup):
    if node.string == '0' or node.string == '1':
        pass
    elif j < i:
        if node.string != '0' and node.string != '1':
            self.preorder(node.left, i, j + 1, nula, jednotka, postup)
            self.preorder(node.right, i, j + 1, nula, jednotka, postup)
    else:
          vypis(self.root)
          print("---------------------------------")
        self.make(node, i, nula, jednotka, postup)
```

Funkcia make dostáva stringy a mení ich na zoznam. Potom zavolá funkciu makeLava na vytvorenie lavej node a makePrava na vytvorenie pravej node.

```
#vytvorenie potomkov
def make(self, node, i, nula, jednotka, postup):
    casti = node.string.split(" + ")

    #vymazanie prazdnych miest
    if "" in casti:
        casti.remove("")

    if len(casti) != 1:
        casti = deMorgan(casti)

    #vymazanie A + A alebo !A + !A
    casti = list(dict.fromkeys(casti))

    #!A + A = 1
    if len(casti) == 2:
        if len(casti[0]) == 2 and len(casti[1]) == 1:
            if casti[0][0] == '!' and casti[0][1] == casti[1][0]:
                node.left = jednotka
                node.right = jednotka
                return

        elif len(casti[0]) == 1 and len(casti[1]) == 2:
            if casti[1][0] == '!' and casti[0][0] == casti[1][1]:
                node.left = jednotka
                node.right = jednotka
                return

    #vytvorenie lavej node
    node.left = Node(makeLava(i, casti, postup))
    if node.left.string == "":
        node.left = nula
    elif node.left.string == "vsetko":
        node.left = jednotka

    #vytvorenie pravej node
    node.right = Node(makePrava(i, casti, postup))
    if node.right.string == "":
        node.right = nula
    elif node.right.string == "vsetko":
        node.right = jednotka
```

Funkcie makeLava a makePrava sú podobné zrkadlové funkcie na vytváranie pravého a lavého prvku. Na vstupe dostanú príklad v tvare zoznamu a vrátia string pre pravý a ľavý node.

```
#vytvorenie stringu pre laveho potomka
def makeLava(k, casti, postup):
    copy = ""
    lava = ""
    for i in range (len (casti)):

        #!ABC prekopiruje BC do lavej node
        if casti[i][0] == '!':
            if casti[i][1] == postup[k]:
                if len(casti[i]) == 2:
                    lava = ""
                    return "vsetko"
                for j in range (len (casti[i]) - 2):
                    copy += casti[i][j + 2]
                lava += copy + " + "
                copy = ""
            else:
                for j in range (len (casti[i])):
                    copy += casti[i][j]
                lava += copy + " + "
                copy = ""

        #A + BC prekopiruje BC do lavej node
        elif casti[i][0] != '!' and casti[i][0] != postup[k]:
            for j in range (len (casti[i])):
                copy += casti[i][j]
            lava += copy + " + "
            copy = ""

    lava = lava.rstrip(" +")
    lava = lava.lstrip(" +")

    #A + A = A alebo !A + !A = !A
    pomoc = lava.split(" + ")
    pomoc = list(dict.fromkeys(pomoc))
    lava = ""
    for i in range(len(pomoc)):
        lava += pomoc[i] + " + "
    lava = lava.rstrip(" +")

    return lava
```

```
#vytvorenie stringu pre praveho potomka
def makePrava(k, casti, postup):
    copy = ""
    prava = ""
    for i in range (len (casti)):

        #ABC prekopiruje BC do pravej node
        if casti[i][0] == postup[k]:
            if len(casti[i]) == 1:
                prava = ""
                return "vsetko"
            else:
                for j in range (len (casti[i]) - 1):
                    copy += casti[i][j + 1]
                prava += copy + " + "
                copy = ""

        #!A + BC prekopiruje BC do pravej node alebo !A + !BC prekopiruje !BC do pravej node
        elif casti[i][0] != '!' or (casti[i][0] == '!' and casti[i][1] != postup[k]):
            for j in range (len (casti[i])):
                copy += casti[i][j]
            prava += copy + " + "
            copy = ""

    prava = prava.rstrip(" +")
    prava = prava.lstrip(" +")

    #A + A = A alebo !A + !A = !A
    pomoc = prava.split(" + ")
    pomoc = list(dict.fromkeys(pomoc))
    prava = ""
    for i in range(len(pomoc)):
        prava += pomoc[i] + " + "
    prava = prava.rstrip(" +")

    return prava
```

Ešte predtým ako sa strom vytvorý, funkcia uprav_vstup upraví používateľov príklad. Napríklad odstráni A.A usporiada premenné podľa toho aký používateľ zvolil postup atd.

```
#upravenie vstupu a zoradenie podla postupu
def uprav_vstup(vstup, postup):
    casti = vstup.split(" + ")
    vstup = ""
    castil = []
    for i in range(len(casti)):
        pomoc = []
        for j in range (len(casti[i])):
            if casti[i][j] in postup:
                if casti[i][j - 1] == '!':
                    pomoc.append(casti[i][j - 1] + casti[i][j])
                else:
                    pomoc.append(casti[i][j])

        #A . A = A alebo !A . !A = !A
        pomoc = list(dict.fromkeys(pomoc))

        #usporiadanie podla postupu
        pomocl = []
        vyraz = ""
        for j in range(len(postup)):
            for k in range (len(pomoc)):
                if pomoc[k][0] == postup[j]:
                    pomocl.append(pomoc[k])
                elif len(pomoc[k]) == 2:
                    if pomoc[k][1] == postup[j]:
                        pomocl.append(pomoc[k])

        for j in range(len(pomocl)):
            vyraz += pomocl[j]

        castil.append(vyraz)

    # !A.A = 0 alebo A!A = 0
    for k in range(len(postup)):
        vymazat1 = '!' + postup[k] + postup[k]
        vymazat2 = postup[k] + '!' + postup[k]
        remove = []
        pomoc = 0
        for j in range (len(castil)):
            if vymazat1 in castil[j]:
                castil[j] = ""
            elif vymazat2 in castil[j]:
                castil[j] = ""

    while "" in castil:
        castil.remove("")

    for i in range (len (castil)):
        vstup += castil[i] + " + "
    vstup = vstup.rstrip(" +")
    print("po uprave: " + vstup)
    return vstup
```

Nakoniec BDD_use testuje vytvorený strom spôsobom, že na základe vstupných jednotiek a núl prechádza stromom. Nakoniec vypíše string prvku v ktorom skončí (malo by byť vždy 1 alebo 0).

```
#testovanie vstupov
def BDD_use(vstup, root, postup):
    node = root
    i = 0
    #1 chod doprava, 0 chod dolava
    while node.right != None and node.left != None:
        if postup[i] in node.string:
            if vstup[i] == '1':
                node = node.right
            elif vstup[i] == '0':
                node = node.left
        i += 1
    print("for " + vstup + " result should be " + node.string)
```

Testovanie korektnosti BDD:

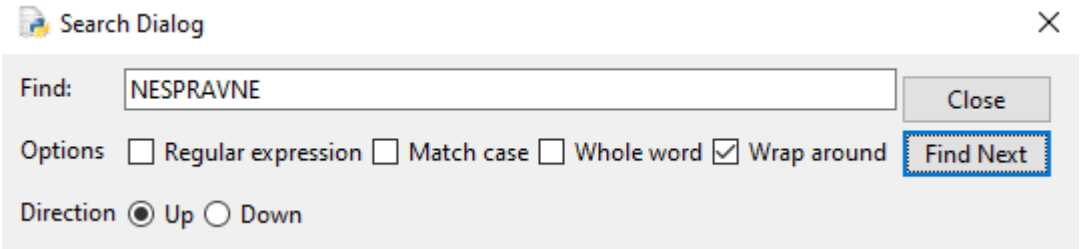
Na testovanie korektnosti výsledkov môjho riešenia som použil funkciu evaluate. Ktorá za samotné premenné príkladu dosadzuje 1 a 0. Počas testovania som zistil, že tento spôsob riešenia booleovských funkcií je veľmi časovo neefektívny.

```
def evaluate(vstup, postup, premenne):
    i = 0
    string = ""
    koniec = ""
    pomoc = uprav_vstup(vstup, postup)
    vysledok1 = True
    for j in range(premenne):
        koniec += "1"
    while string != koniec:
        vstup = pomoc
        string = bin(i)
        string = string.lstrip("0b")
        for j in range(premenne - len(string)):
            string = "0" + string
        for j in range(premenne):
            vstup = vstup.replace(postup[j], string[j])

        for j in range(len(vstup)):
            if vstup[j] == '!':
                if vstup[j + 1] == '1':
                    temp = list(vstup)
                    temp[j + 1] = '0'
                    vstup = "".join(temp)
                elif vstup[j + 1] == '0':
                    temp = list(vstup)
                    temp[j + 1] = '1'
                    vstup = "".join(temp)
        vstup = vstup.replace('!', '')
        casti = vstup.split(" + ")
        vysledok = '0'
        for j in range(len (casti)):
            if not "0" in casti[j]:
                vysledok = '1'
                break
        if BDD_use(string, tree.root, postup) != vysledok:
            vysledok1 = False
            return vysledok

        i += 1
    return vysledok1
```

Testovanie pri 4 premenných. (ABCD, 100 náhodne vygenerovaných príkladov)

```
!A!A!A + !C!B + !CBD + !B!AD!C
po uprave: !A + !B!C + B!CD + !A!B!CD
POCET PRVKOV:
7
REDUKCIA:
77.4%
po uprave: !A + !B!C + B!CD + !A!B!CD
SPRAVNE
```

Na obrázku je príklad výpisu. Percentuálna miera redukcie je v tomto prípade 77.4%.



Po prehľadaní celej konzoly nenašlo ani jeden nesprávny výpis „NESPRAVNE“.

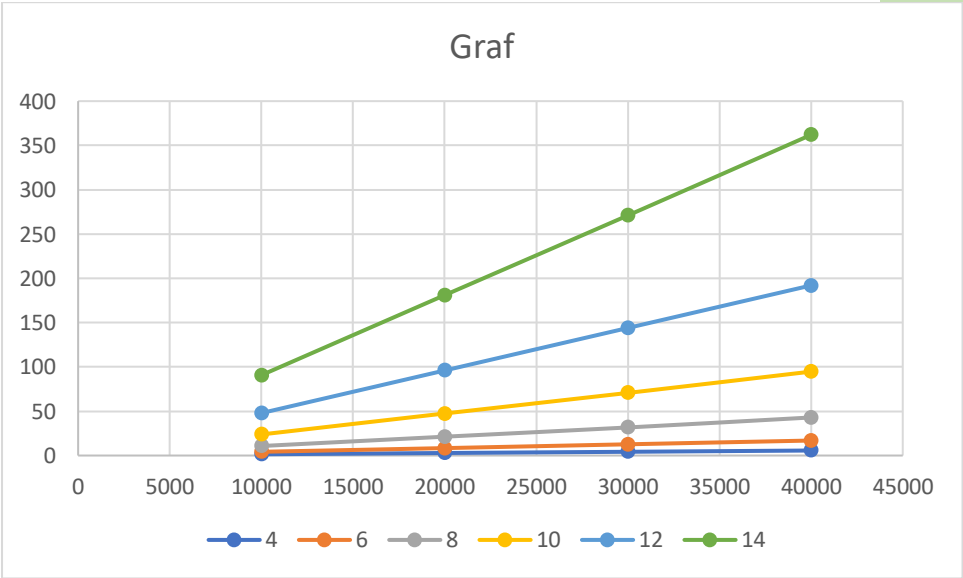
Podobne som testoval aj pre 8 premenných (QWERTYUI), 12 premenných (LKJHGFDSATRE) a 16 premenných (MNBVCXZASDFGHJKL) ani v jednom z testov nevyšiel nesprávny výsledok.

Percentuálna miera redukcie pri 16 premenných dosahovala v priemere 99.95% .

Testovanie času vytvorenia BDD:

Časovú náročnosť som testoval bez vypisovania korektnosti alebo samotných stromov. Hodnoty v tabuľke zvýraznené zelenou som intuitívne doplnil.

počet vytvorení/premenné	4	6	8	10	12	14
10000	1,405	4,203	10,7	23,777	47,903	90,703
20000	2,86	8,368	21,265	47,455	95,993	181
30000	4,214	12,679	31,668	70,653	144	271
40000	5,651	16,911	42,951	94,739	192	362



Na základe grafu a tabuľky je môj odhad časovej náročnosti O(n*log n).

Pamäťová náročnosť a miera redukcie:

Na vypočítanie miery redukcie je potrebný počet nodov kompletného neredukovaného diagramu a počet zredukovaných/odstránených nodov. Počet nodov vo vytvorenom, zredukovanom diagrame zisťuje funkcia count_nodes.

```
def count_nodes(self, node, zoznam):
    if node is None:
        return 0
    self.count_nodes(node.left, zoznam)
    if node not in zoznam:
        zoznam.append(node)
    self.count_nodes(node.right, zoznam)
    return zoznam
```

Počet nodov v neredukovanom diagrame:

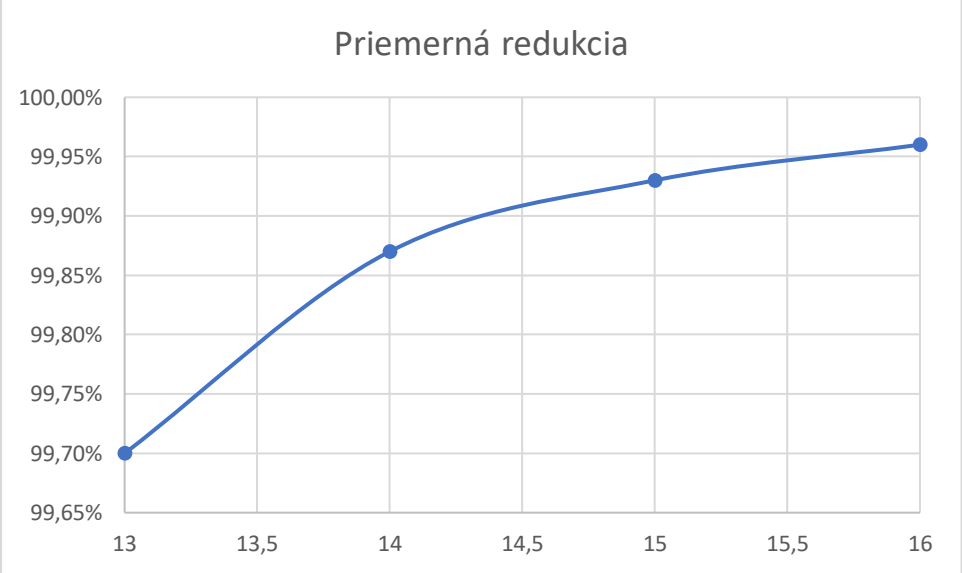
```
neredukovany_pocet = (2 ** (len(postup) + 1)) - 1
```

Následne sa miera redukcie vypočíta jednoduchým vzorcom.

```
print(str(round((((neredukovany_pocet - tree.pocet) / neredukovany_pocet) * 100), 1)) + "%")
```

Mieru redukcie som testoval vytvorením 100 diagramov pri 13, 14, 15 a 16 premenných. Pri priemernom počte nodov v diagramoch sa vyskytli výnimky, pravdebodobne kvôli funkcii na náhodné vytváranie výrazov, avšak priemernú redukciu to veľmi neovplyvnilo. Priemerná redukcia narastala logaritmicky.

	Priemerný počet nodov	Priemerná redukcia
13	40,5	99,70%
14	41,8	99,87%
15	48,9	99,93%
16	44,95	99,96%



Moje testovanie malo tak vysokú mieru redukcie pravdepodobne kvôli tomu, že randomne vygenerovaný výraz sa dal veľmi efektívne upraviť.

Keďže aj pri 16 premenných sa počet nodov pohybuje okolo 45, pamäťová náročnosť je veľmi nízka. A taktiež rastie logaritmicky.