

به نام خدا

حدیث غفوری ۹۸۲۵۴۱۳

پروژه ی درس هوش مصنوعی

الگوریتم (Davis–Putnam–Logemann–Loveland (DPLL

```
solve_dpll(cnf):
    while(cnf has a unit clause {X}):
        delete clauses containing {X}
        delete {!X} from all clauses
    if null clause exists:
        return False
    if CNF is null:
        return True
    select a literal {X}
    cnf1 = cnf + {X}
    cnf2 = cnf + {!X}
    return solve_dpll(cnf1)+solve_dpll(cnf2)
```

توضیح الگوریتم

این الگوریتم کامل است و براساس منطق backtracking عمل میکند.

در این الگوریتم از یک ورودی به فرمت CNF استفاده میشود و تابع حل DPLL به صورت بازگشتی فراخوانی میشود.

به کلاوزی یک کلاوز واحد میگوییم که از یک سمبل یا ترکیب سمبل و ! تشکیل شده باشد.

تا زمانی که یک کلاوز واحد داریم میتوانیم آن را از بقیه ی کلاوز های ورودی حذف کنیم.

اگر خود کلاوز تشخیص داده شده به صورت دقیق در کلاوز دیگری تکرار شده باشد، آن کلاوز را به طور کلی حذف میکنیم و اگر نقیض آن سمبل در کلاوز دیگری تکرار شده باشد فقط همان سمبل را حذف میکنیم. به این کار عملیات unit propagation میگوییم.

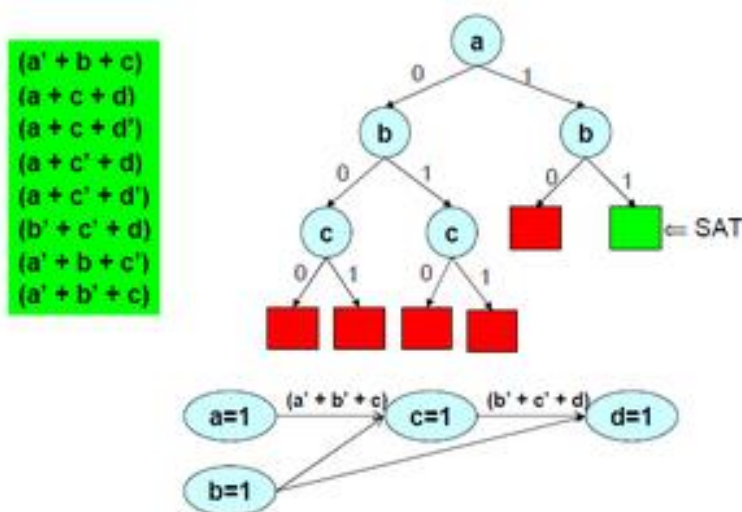
پس از این عملیات باید دو بررسی انجام دهیم.

۱. اگر کلاوزی داریم که تهی است باید در خروجی False برگردانیم به معنای اینکه با این مقداردهی نمیتوانیم مساله را ارضا کنیم.

۲. اگر CNF ورودی که داریم مدام اپدیت میکنیم در نهایت تهی شود یعنی همه ی کلاوزها به درستی مقداردهی و ارضاشده اند پس True برمیگردانیم.

در غیر این دو حالت باید شاخه های جدید درخت را بسازیم و به متغیر یا لیترال جدیدی مقدار دهیم پس باید یک سمبل از لیست سمبل های موجود را انتخاب کنیم و به کل CNF ورودی اضافه کنیم و اول با مقداردهی TRUE ببینیم به CONFLICT ای میرسیم یا نه؟ اگر رسیدیم باید BACKTRACK کنیم و به جای TRUE باید FALSE برای آن متغیر در نظر بگیریم و با این مقداردهی باید بار دیگر ارضاپذیری متغیرها را بررسی کنیم.

هربار که به یک CONFLICT برسیم یعنی به مقداردهی ای برسیم که باعث نقض ارضا پذیری متغیرها شود باید عقبگرد کنیم و از شاخه ی درخت بالارویم و مقدار دیگری را بررسی کنیم.



به طور مثال در شکل بالا ابتدا دنبال unit clause ها میگردیم و چون نداریم باید از یک سمبل از لیست سمبل ها انتخاب کنیم و شروع به مقدار دهی کنیم.

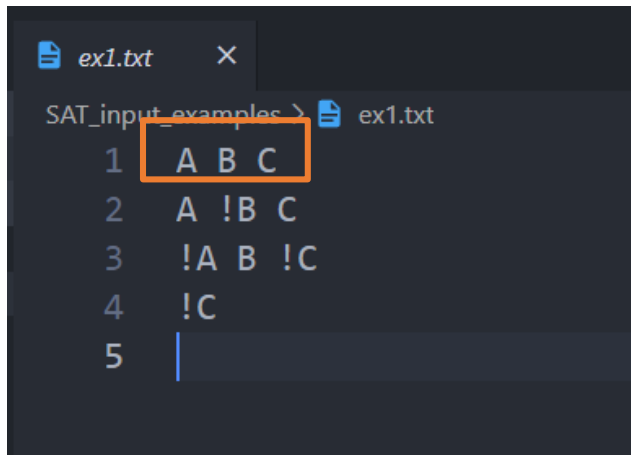
در اینجا ابتدا به سمبل a مقدار False یا صفر می دهیم و سپس به b صفر می دهیم و در نهایت به c که صفر می دهیم به تناقض یا کانفلیکت میرسیم.

پس باید به c مقدار دیگری یعنی یک یا true بدهیم و ارضاپذیری کل مساله را چک کنیم که باز هم به کانفلیکت میرسیم. پس باید یک لول از درخت بالا رویم یعنی به سراغ مقدار دهی b که قبلا دیدیم با مقدار False به کانفلیکت میرسد پس این بار شاخه ی جدید درخت را امتحان میکنیم و مقدار true می دهیم.

به همین ترتیب شاخه های درخت را چک میکنیم و اگر به کانفلیکت رسیدیم به عقب برمیگردیم و مقدار دیگری را انتخاب میکنیم. تا زمانی که همه ی شرط ها ارضا شوند.

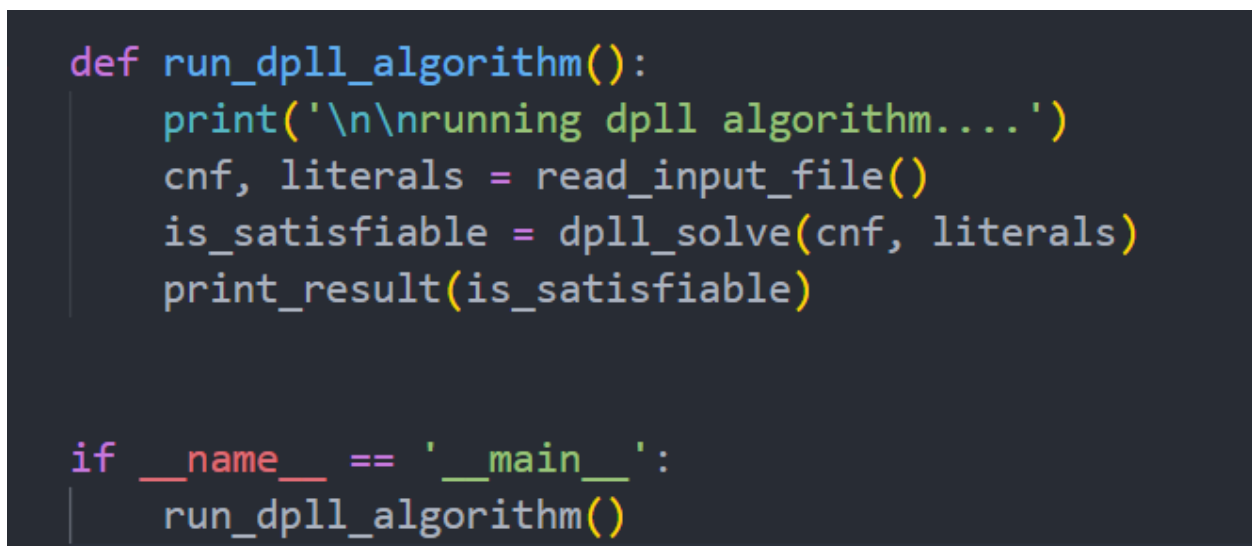
ورودی برنامه

ورودی باید به فرمت زیر و در یک فایل باشد از انجایی که فرمت ما CNF است باید در هر خط از فایل یک CLAUSE قرار دهیم که هر کدام از این CLAUSE ها شامل یک یا چندین LITERAL هستند که با هم OR شده اند و در نهایت این CLAUSE ها با هم AND میشوند.



```
ex1.txt x
SAT_input_examples > ex1.txt
1 A B C
2 A !B C
3 !A B !C
4 !C
5
```

تابع اصلی برنامه



```
def run_dp11_algorithm():
    print('\n\nrunning dp11 algorithm....')
    cnf, literals = read_input_file()
    is_satisfiable = dp11_solve(cnf, literals)
    print_result(is_satisfiable)

if __name__ == '__main__':
    run_dp11_algorithm()
```

در حالت main این تابع فراخوانی میشود که سه وظیفه دارد.

۱. خواندن فایل ورودی و ذخیره ی cnf,literals
۲. اجرای الگوریتم dp11
۳. چاپ نتایج (ارضایپذیری یا غیرقابل ارضا)

کد خواندن ورودی

```
def read_input_file():  
    input_file = open(sys.argv[1], 'r').read()  
    cnf = input_file.splitlines()  
    literals = get_alphabet_literals(cnf)  
    print("cnf: ", cnf)  
    print("literals: ", literals)  
    return cnf, literals
```

در این تابع نام فایلی که می‌خواهیم CNF را از آن بخوانیم از CMD می‌گیریم و به کمک تابع `splitlines` فایل را به خط‌هایی که دارد تبدیل می‌کنیم و در یک آرایه به نام `cnf` ذخیره می‌کنیم.

پس متغیر `cnf` شامل لیستی از کلاوزهای ورودی است.

لیست سمبل‌ها را به کمک تابع `get_alphabet_literals` می‌گیریم و در متغیر `literals` ذخیره می‌کنیم و در نهایت این مقادیر را چاپ می‌کنیم و به عنوان خروجی تابع برمی‌گردانیم.

تابع دریافت سمبل‌ها

```
def get_alphabet_literals(cnf):  
    alphabet_literals_list = []  
    cnf_str = ''.join(cnf)  
    for alphabet in list(set(cnf_str)):  
        if alphabet.isalpha():  
            alphabet_literals_list.append(alphabet)  
    return alphabet_literals_list
```

در این تابع به عنوان ورودی `cnf` را می‌گیریم و یک حلقه فور روی آن می‌زنیم تا سمبل‌ها را شناسایی کنیم.

به کمک تابع `isalpha` بررسی می‌کنیم که سمبل انتخاب شده جزو حروف الفبا باشد.

در نهایت لیست سمبل‌ها را برمی‌گردانیم.

تابع چاپ cnf

```
def print_CNF_form(cnf):
    message = ''
    for clause in cnf:
        if len(clause):
            message += '(' + clause.replace(' ', '+') + ')'
    if message == '':
        message = '()'
    print('cnf = '+message)
```

از انجایی که cnf در فایل ورودی به شکل واضح وجود ندارد زمانی که میخواهیم فرم دقیق آن را ببینیم از این تابع استفاده میکنیم. در هر کلاوز بین لیترال ها علامت + و کل کلاوز را در پرانتز نمایش میدهیم اگر هم cnf تهی یا خالی باشد یک پرانتز خالی نمایش میدهیم.

تابع نمایش نتیجه نهایی

```
def print_result(is_satisfiable):
    global true_value_set, false_value_set
    print(f"""
    number of unit propagations:{num_propagations}
    number of splits in tree:{num_splits}
    """)

    if is_satisfiable:
        print('SAT problem is satisfiable\nSolution of Problem is ')
        for literal in true_value_set:
            print(f'{literal} = TRUE')

        for literal in false_value_set:
            print(f'{literal} = FALSE')

    else:
        print('***** SAT problem is not satisfiable *****')
```

در این تابع براساس بولین is_satisfiable تصمیم میگیریم که چاپ خروجی چگونه باشد.

در برنامه دو متغیر گلوبال به نام `num_splits` و `num_propagations` داریم که اولی نشان میدهد چندبار عملیات `unit propagation` انجام میشود و دومی نشان میدهد چندبار از درخت پایین میرویم و دو شاخه ی جدید ایجاد میشود. در این تابع این مقادیر چاپ میشوند و اگر مسئله ارضا پذیر باشد مقادیرهای انتساب شده چاپ میشوند و اگر نباشد میگوییم ارضا پذیر نیست.

تابع اصلی حل dpll

```
def dpll_solve(cnf, alphabet_literals):
    global num_splits, num_propagations, true_value_set, false_value_set
    num_splits += 1
    print_CNF_form(cnf)
    temp_true_set = []
    temp_false_set = []
    true_value_set = set(true_value_set)
    false_value_set = set(false_value_set)
    cnf = list(set(cnf))
    units = find_units(cnf)
    print('Units =', units)
    for unit in units:
        num_propagations += 1
        if '!' in unit:
            temp_false_set.append(unit[-1])
            cnf = handle_unit_propagation_false(cnf, unit)
        else:
            temp_true_set.append(unit)
            cnf = handle_unit_propagation_true(cnf, unit)
    print('\nCNF after calling unit propagation = ', cnf, '\n')
```

در این تابع به ازای هربار فراخوانی، یک سطح در درخت پایین میرویم پس مقدار `num_splits` یکی اضافه میشود.

در هر مرحله مقدار متغیر `cnf` را به صورت بیان شده در تابع بالا، مینویسیم.

برای هر کدام از مقادیر `true`, `false` دو متغیر داریم به نام های `temp_true_set`, `temp_false_set` این مقادیر موقتی در هر مرحله از درخت هستند که به متغیرها اختصاص میدهیم و اگر به کانفلیکت خوردیم مقادیر موجود در این ارایه را از ارایه ی اصلی مقادیر درست و نادرست پاک میکنیم.

دو مجموعه (`set`) به نام های `true_value_set`, `false_value_set` داریم که دلیل انتخاب مجموعه، عدم تکرار مقادیر موجود در آن است.

در این متغیرها که در هر مرحله هم اپدیت میشوند، مقادیر انتساب شده به سمبل ها یا متغیرهای مسئله را قرار میدهیم.

برای پیدا کردن unit ها از تابع find_units() استفاده میکنیم و باید برای هر unit عملیات unit propagation را انجام دهیم. پس به ازای هر unit یک مقدار به تعداد num_propagations اضافه میشود. چون داریم به صورت موقتی مقادیر را انتساب میکنیم در ارایه های Temp گفته شده ذخیره میکنیم. در اینجا اگر لیترال مثبت داشته باشیم تابع handle_unit_propagation_true(cnf, unit) و اگر لیترال منفی داشته باشیم تابع handle_unit_propagation_false(cnf, unit) را فراخوانی میکنیم. در نهایت پس از اتمام این عملیات باید cnf اپدیت شده را چاپ کنیم.

```
if len(cnf) == 0:
    return True

if is_empty_clause(cnf):
    handle_backtrack(temp_true_set, temp_false_set)
    return False

alphabet_literals = get_alphabet_literals(cnf)
selected_literal = alphabet_literals[0]
if dpll_solve(deepcopy(cnf)+[selected_literal], deepcopy(alphabet_literals)):
    return True
elif dpll_solve(deepcopy(cnf)+['!'+selected_literal], deepcopy(alphabet_literals)):
    return True
else:
    remove_assigned_values(temp_true_set, temp_false_set)
    return False
```

در ادامه اگر طول متغیر cnf که یک ارایه است صفر باشد یعنی هیچ عضو دیگری ندارد و یعنی تهی است پس مقدار True را برمیگردانیم یعنی ارضا پذیر بوده ان انتساب اگر یک کلاوز تهی پیدا کنیم باید عقبگرد کنیم و مقدار انتساب شده را حذف کنیم و مقدار جدید انتساب کنیم بنابراین False برمیگردانیم.

در غیراین دو صورت هم لیترال جدیدی انتخاب میکنیم و مقدار درست و نادرست را جداگانه به ان انتساب میکنیم و هر کدام را بررسی میکنیم اگر یکی از ان درست باشد که ادامه میدهیم اگر هیچ کدام درست نباشد باید در درخت یک سطح بالا رویم و متغیر پدر که باعث ساخت این دو فرزند شده را فلیپ کنیم یعنی اگر درست بوده نادرست کنیم و اگر نادرست بوده درست کنیم.

تابع هندل کردن عملیات unit propagation برای مقادیر نادرست

```
def handle_unit_propagation_false(cnf, unit): # cnf is an array
    global false_value_set
    alphabet_literal = unit[-1]
    false_value_set.add(alphabet_literal)
    # loop through all clauses to find this literal or its ~
    clause_counter = 0
    while True:
        if unit in cnf[clause_counter]:
            # delete clause with this literal in it
            cnf.remove(cnf[clause_counter])
            clause_counter -= 1
        elif alphabet_literal in cnf[clause_counter]:
            cnf[clause_counter] = cnf[clause_counter].replace(
                alphabet_literal, '').strip()
            clause_counter += 1
        if clause_counter >= len(cnf):
            break
    return cnf
```

مجموعه های false_value_set, true_value_set به صورت سراسری و گلوبال تعریف شده اند پس در این تابع باید از کلمه ی global استفاده کنیم.

سمبل یا الفبای لیترال را به کمک unit[-1] که عضو اخر ارایه را استخراج میکند بدست می اوریم و این مقدار را به مجموعه ی false_value_set اضافه میکنیم (اگر در آینده باعث ایجاد کانفلیکت شود حذف میشود)

به ازای تمام کلاوز های موجود در صورت سوال باید تک تک انها را بررسی کنیم که ایا این سمبل یا نقیض ان را دارند یا خیر؟

اگر نقیض ان را داشتند که کل ان کلاوز حذف میشود چون نقیض ان سمبل را درست درنظر گرفتیم و چون میدانیم بین لیترال ها در یک کلاوز عملگر or قرار دارد پس کل کلاوز درست میشود پس از remove() استفاده میکنیم و ان را حذف میکنیم از cnf و مقدار کانترا را یکی کم میکنیم چون تعداد کلاوز های یکی کم شد.

اگر دقیقا خود سمبل بدون علامت نقیض وجود داشته باشد باید فقط همان سمبل را حذف کنیم و بقیه ی لیترال های موجود در ان کلاوز دست نخورده میمانند پس آن سمبل را با " یعنی با استرینگ تهی

جایگزین میکنیم تا حذف شود.
در نهایت زمانی که اندیس در این حلقه بیشتر از تعداد کلاوز ها شود از حلقه خارج میشویم.

تابع هندل کردن عملیات unit propagation برای مقادیر نادرست

```
5 def handle_unit_propagation_true(cnf, unit):
6     global true_value_set
7     true_value_set.add(unit)
8     clause_counter = 0
9     while True:
10         if '!' + unit in cnf[clause_counter]:
11             cnf[clause_counter] = cnf[clause_counter].replace(
12                 '!' + unit, '').strip()
13             if ' ' in cnf[clause_counter]:
14                 cnf[clause_counter] = cnf[clause_counter].replace(' ', '')
15         elif unit in cnf[clause_counter]:
16             cnf.remove(cnf[clause_counter])
17             clause_counter -= 1
18         clause_counter += 1
19         if clause_counter >= len(cnf):
20             break
21     return cnf
```

این تابع هم مشابه تابع بالایی است با این تفاوت که برای مقادیر درست کار میکند و اگر علامت نقیض در کنار سمبل باشد در یک کلاوز، فقط همان سمبل و نقیض آن حذف میشوند و اگر علامت ! نباشد و سمبل به تنهایی باشد (لیترال مثبت) کل آن کلاوز های شامل این سمبل حذف میشوند.

تابع تشخیص کلاوز تهی در cnf

```
def is_empty_clause(cnf):
    counter = 0
    for clause in cnf:
        if len(clause) == 0:
            counter += 1
    if counter > 0:
        return True
    return False
```

در این تابع اگر حتی یک کلاوز هم داشته باشیم که تهی باشد، مقدار درست را برمیگردانیم تا بتوانیم عملیات عقبگرد یا backtrack را انجام دهیم.

تابع هندل کردن عقب نشینی

```
def handle_backtrack(temp_true_set, temp_false_set):  
    global true_value_set, false_value_set  
    for alphabet_literal in temp_true_set:  
        true_value_set.remove(alphabet_literal)  
    for alphabet_literal in temp_false_set:  
        false_value_set.remove(alphabet_literal)  
    print('##### Null clause found, backtracking occurred #####')
```

در این تابع چون می خواهیم از درخت بالارویم باید مقادیر انتساب داده شده را پاک کنیم پس تمام مقادیری که در ارایه های تمپ هستند را از مجموعه های درست و نادرست اصلی، حذف میکنیم.

خروجی برنامه

```
Win 10@whoami MINGW64 ~/Desktop/university/term7/AI/hadis_project/hadis_ghafouri_9825413  
$ python dpll_SAT_implementation.py SAT_input_examples/ex1.txt  
  
running dpll algorithm....  
cnf: ['A B C', 'A !B C', '!A B !C', '!C']  
literals: ['B', 'A', 'C']  
cnf = (A+B+C)(A+!B+C)(!A+B+!C)(!C)  
Units = ['!C']  
  
CNF after calling unit propogation = ['A !B', 'A B']  
  
cnf = (A+!B)(A+B)(B)  
Units = ['B']  
  
CNF after calling unit propogation = ['A']  
  
cnf = (A)(A)  
Units = ['A']  
  
CNF after calling unit propogation = []  
  
number of unit propagations:3  
number of splits in tree:3  
  
SAT problem is satisfiable  
Solution of Problem is  
B = TRUE  
A = TRUE  
C = FALSE
```

همان گونه که مشاهده میکنید این مساله ارضا پذیر است با مقادیر منتسب شده به صورت زیر:

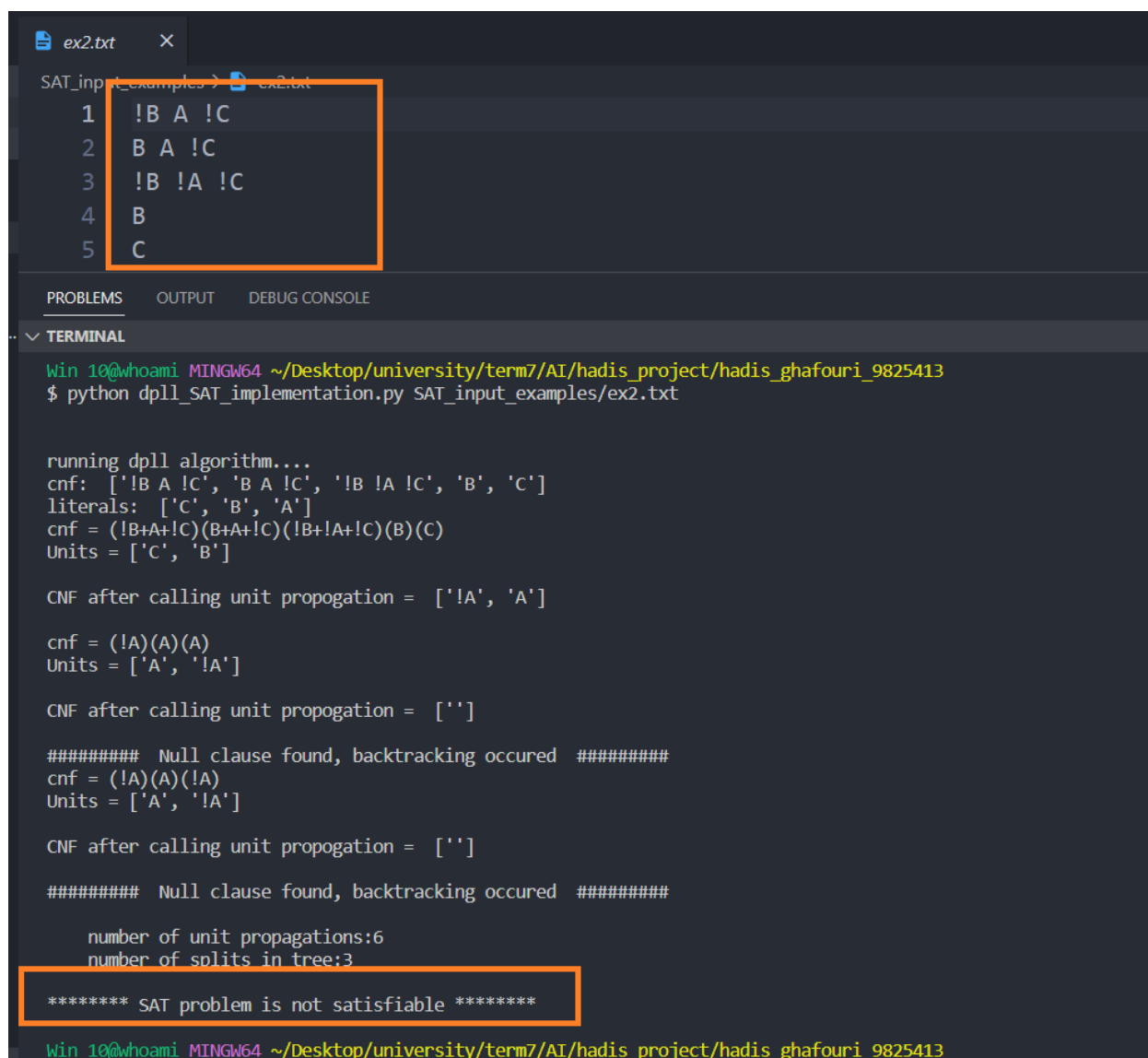
A = TRUE

B = TRUE

C = FALSE

مثال دو

به ازای ورودی Ex2.txt، مساله ی داده شده ارضا پذیر نیست.



The screenshot shows a code editor with a file named `ex2.txt` open. The file contains five lines of SAT input:

```
1 !B A !C
2 B A !C
3 !B !A !C
4 B
5 C
```

The terminal output shows the execution of the `python dpll_SAT_implementation.py SAT_input_examples/ex2.txt` command. The output displays the internal state of the SAT solver, including the current clause list (`cnf`), literals, and units. It shows the process of unit propagation and backtracking, ultimately concluding that the SAT problem is not satisfiable.

```
Win 10@whoami MINGW64 ~/Desktop/university/term7/AI/hadis_project/hadis_ghafouri_9825413
$ python dpll_SAT_implementation.py SAT_input_examples/ex2.txt

running dpll algorithm....
cnf: ['!B A !C', 'B A !C', '!B !A !C', 'B', 'C']
literals: ['C', 'B', 'A']
cnf = (!B+A!C)(B+A!C)(!B+!A!C)(B)(C)
Units = ['C', 'B']

CNF after calling unit propogation = ['!A', 'A']

cnf = (!A)(A)(A)
Units = ['A', '!A']

CNF after calling unit propogation = []

##### Null clause found, backtracking occured #####
cnf = (!A)(A)(!A)
Units = ['A', '!A']

CNF after calling unit propogation = []

##### Null clause found, backtracking occured #####

number of unit propagations:6
number of splits in tree:3

***** SAT problem is not satisfiable *****

Win 10@whoami MINGW64 ~/Desktop/university/term7/AI/hadis_project/hadis_ghafouri_9825413
```

