



**Hacettepe University**  
**Department of Computer Engineering**

---

**BBM405**  
**Fundamentals of Artificial Intelligence**

Eylül Tuncel  
21727801

BBM405  
Fundamentals of Artificial Intelligence  
**Spring 2021 - Homework 2**

**1. Introduction**

In this homework, we were expected to solve different constraint satisfaction problems with different approaches. One of the approach is solving with theorem provers like z3 or some constraint satisfaction algorithms. I chose a backtracking search algorithm for CSP problems.

**2. Part 1 : Write 5 propositional formulas in CNF and find the satisfying values.**

2.1. Propositional Formula 1:

$$(a \vee b) \wedge (b \vee c) \wedge (c \vee d) \wedge (d \vee a) \wedge (a \vee c) \wedge (b \vee \neg c) \wedge (c \vee \neg d) \wedge (d \vee b)$$

You can see the values that satisfy the above propositional formula in the truth table. And below that you can see the propositional formula solved by one of theorem provers **z3**.

a	b	c	d	$((a \vee b) \wedge ((b \vee c) \wedge ((c \vee d) \wedge ((d \vee a) \wedge ((a \vee c) \wedge ((b \vee \neg c) \wedge ((c \vee \neg d) \wedge (d \vee b))))))))$
F	T	T	T	T
T	T	T	F	T
T	T	T	T	T

```
3SAT_1.smt
~/Desktop/BBM405/hw2
Open Save
(set-logic QF_UF)
(set-option :produce-models true)
(declare-const A Bool)
(declare-const B Bool)
(declare-const C Bool)
(declare-const D Bool)
(assert
  (and
    (or A B )
    (or B C )
    (or C D )
    (or D A )
    (or A C )
    (or B (not C) )
    (or C (not D) )
    (or D B )
  )
)
(check-sat)
(get-model)
(exit)
```

The model found by z3 is one of the satisfying values of the propositional formula. In the first line you can see that z3 checks for satisfiability and then finds it satisfiable. After that creates a model that has one instance of satisfying values. The values are also seen in the first line of the truth table.

```
eylul@kia:~/Desktop/BBM405/hw2$ z3 -smt2 3SAT_1.smt
sat
(
  (define-fun A () Bool
    false)
  (define-fun D () Bool
    true)
  (define-fun B () Bool
    true)
  (define-fun C () Bool
    true)
)
```

## 2.2. Propositional Formula 2 :

$$(a \vee b \vee \neg c) \wedge (b \vee c \vee \neg d) \wedge (c \vee d \vee \neg a) \wedge (\neg a \vee \neg b \vee \neg d) \wedge (b \vee \neg c \vee \neg d)$$

You can see the values that satisfy the above propositional formula in the truth table. And below that you can see the propositional formula solved by one of theorem provers **z3**.

a	b	c	d	$(a \vee b \vee \neg c) \wedge (b \vee c \vee \neg d) \wedge (c \vee d \vee \neg a) \wedge (\neg a \vee \neg b \vee \neg d) \wedge (b \vee \neg c \vee \neg d)$
F	F	F	F	T
F	T	F	F	T
F	T	F	T	T
F	T	T	F	T
F	T	T	T	T
T	F	T	F	T
T	T	T	F	T

```

Open 3SAT_2.smt Save
~/Desktop/BBM405/hw2
(set-logic QF_UF)
(set-option :produce-models true)
(declare-const A Bool)
(declare-const B Bool)
(declare-const C Bool)
(declare-const D Bool)
(assert
  (and
    (or A B (not C) )
    (or B C (not D) )
    (or C D (not A) )
    (or (not A) (not B) (not D) )
    (or B (not C) (not D) )
  )
)
(check-sat)
(get-model)
(exit)
```

The model found by z3 is one of the satisfying values of the propositional formula. In the first line you can see that z3 checks for satisfiability and then finds it satisfiable. After that creates a model that has one instance of satisfying values. The values are also seen in the fifth line of the truth table.

```
eylul@kia:~/Desktop/BBM405/hw2$ z3 -smt2 3SAT_2.smt
sat
(
  (define-fun A () Bool
    false)
  (define-fun D () Bool
    true)
  (define-fun B () Bool
    true)
  (define-fun C () Bool
    true)
)
```

### 2.3. Propositional Formula 3:

$$(a \vee b \vee \neg c) \wedge (\neg b \vee d \vee e) \wedge (\neg a \vee b \vee c) \wedge (b \vee d \vee \neg e)$$

You can see the values that satisfy the above propositional formula in the truth table. And below that you can see the propositional formula solved by one of theorem provers **z3**.

a	b	c	d	e	$(a \vee b \vee \neg c) \wedge (\neg b \vee d \vee e) \wedge (\neg a \vee b \vee c) \wedge (b \vee d \vee \neg e)$
F	F	F	F	F	T
F	F	F	T	F	T
F	F	F	T	T	T
F	T	F	F	T	T
F	T	F	T	F	T
F	T	F	T	T	T
F	T	T	F	T	T
F	T	T	T	F	T
F	T	T	T	T	T
T	F	T	F	F	T
T	F	T	T	F	T
T	F	T	T	T	T
T	T	F	F	T	T
T	T	F	T	F	T
T	T	F	T	T	T
T	T	T	F	T	T
T	T	T	T	F	T
T	T	T	T	T	T

```

3SAT_3.smt
~/Desktop/BBM405/hw2
Open Save
(set-logic QF_UF)
(set-option :produce-models true)
(declare-const A Bool)
(declare-const B Bool)
(declare-const C Bool)
(declare-const D Bool)
(declare-const E Bool)
(assert
  (and
    (or A B (not C) )
    (or (not B) D E )
    (or (not A) B C )
    (or B D (not E) )
  )
)
(check-sat)
(get-model)
(exit)

```

The model found by z3 is one of the satisfying values of the propositional formula. In the first line you can see that z3 checks for satisfiability and then finds it satisfiable. After that creates a model that has one instance of satisfying values. The values are also seen in the first line of the truth table.

```

eylul@kia:~/Desktop/BBM405/hw2$ z3 -smt2 3SAT_3.smt
sat
(
  (define-fun A () Bool
    false)
  (define-fun D () Bool
    false)
  (define-fun B () Bool
    false)
  (define-fun C () Bool
    false)
  (define-fun E () Bool
    false)
)

```

2.4. Propositional Formula :

$$(\neg a \vee b) \wedge (\neg b \vee c) \wedge (\neg c \vee d) \wedge (\neg d \vee e) \wedge (a \vee \neg b) \wedge (b \vee \neg c) \wedge (c \vee \neg d) \wedge (d \vee \neg e)$$

You can see the values that satisfy the above propositional formula in the truth table. And below that you can see the propositional formula solved by one of theorem provers **z3**.

a	b	c	d	e	$(\neg a \vee b) \wedge (\neg b \vee c) \wedge (\neg c \vee d) \wedge (\neg d \vee e) \wedge (a \vee \neg b) \wedge (b \vee \neg c) \wedge (c \vee \neg d) \wedge (d \vee \neg e)$
F	F	F	F	F	T
T	T	T	T	T	T

```

3SAT_4.smt
~/Desktop/BBM405/hw2
Open Save
(set-logic QF_UF)
(set-option :produce-models true)
(declare-const A Bool)
(declare-const B Bool)
(declare-const C Bool)
(declare-const D Bool)
(declare-const E Bool)
(assert
  (and
    (or (not A) B )
    (or (not B) C )
    (or (not C) D )
    (or (not D) E )
    (or A (not B) )
    (or B (not C) )
    (or C (not D) )
    (or D (not E) )
  )
)
(check-sat)
(get-model)
(exit)

```

The model found by z3 is one of the satisfying values of the propositional formula. In the first line you can see that z3 checks for satisfiability and then finds it satisfiable. After that creates a model that has one instance of satisfying values. The values are also seen in the first line of the truth table.

```

eylul@kia:~/Desktop/BBM405/hw2$ z3 -smt2 3SAT_4.smt
sat
(
  (define-fun A () Bool
    false)
  (define-fun D () Bool
    false)
  (define-fun B () Bool
    false)
  (define-fun C () Bool
    false)
  (define-fun E () Bool
    false)
)

```

2.5. Propositional Formula :

$$\begin{aligned}
 & (e \vee y \vee l) \wedge (l \vee u \vee l) \wedge (t \vee u \vee n) \wedge (c \vee e \vee l) \wedge \\
 & (-e \vee -y \vee l) \wedge (-l \vee -u \vee l) \wedge (-t \vee -u \vee n) \wedge (-c \vee -e \vee l)
 \end{aligned}$$

You can see the values that satisfy the above propositional formula in the truth table. And below that you can see the propositional formula solved by one of theorem provers **z3**.

c	e	l	n	t	u	y	$(e \vee y \vee l) \wedge (l \vee u \vee l) \wedge (t \vee u \vee n) \wedge (c \vee e \vee l) \wedge (\neg e \vee \neg y \vee l) \wedge (\neg l \vee \neg u \vee l) \wedge (\neg t \vee \neg u \vee n) \wedge (\neg c \vee \neg e \vee l)$
F	F	T	F	F	T	F	T
F	F	T	F	F	T	T	T
F	F	T	F	T	F	F	T
F	F	T	F	T	F	T	T
F	F	T	T	F	F	F	T
F	F	T	T	F	F	T	T
F	F	T	T	F	T	F	T
F	F	T	T	F	T	T	T
F	F	T	T	T	F	F	T
F	F	T	T	T	F	T	T
F	F	T	T	T	T	F	T
F	F	T	T	T	T	T	T
F	T	F	F	F	T	F	T
F	T	F	T	F	T	F	T
F	T	F	T	T	T	F	T
F	T	T	F	F	T	F	T
F	T	T	F	F	T	T	T
F	T	T	F	T	F	F	T
F	T	T	F	T	F	T	T
F	T	T	T	F	F	F	T
F	T	T	T	F	F	T	T
F	T	T	T	F	T	F	T
F	T	T	T	F	T	T	T
F	T	T	T	T	F	F	T
F	T	T	T	T	F	T	T
F	T	T	T	T	T	F	T

F	T	T	T	T	T	T	T
T	F	F	F	F	T	T	T
T	F	F	T	F	T	T	T
T	F	F	T	T	T	T	T
T	F	T	F	F	T	F	T
T	F	T	F	F	T	T	T
T	F	T	F	T	F	F	T
T	F	T	F	T	F	T	T
T	F	T	T	F	F	F	T
T	F	T	T	F	F	T	T
T	F	T	T	F	T	F	T
T	F	T	T	F	T	T	T
T	F	T	T	T	F	F	T
T	F	T	T	T	F	T	T
T	F	T	T	T	T	F	T
T	F	T	T	T	T	T	T
T	T	T	F	F	T	F	T
T	T	T	F	F	T	T	T
T	T	T	F	T	F	F	T
T	T	T	F	T	F	T	T
T	T	T	T	F	F	F	T
T	T	T	T	F	F	T	T
T	T	T	T	F	T	F	T
T	T	T	T	F	T	T	T
T	T	T	T	T	F	F	T
T	T	T	T	T	F	T	T
T	T	T	T	T	T	F	T
T	T	T	T	T	T	T	T



```
3SAT_5.smt
~/Desktop/BBM405/hw2
Open Save

(set-logic QF_UF)
(set-option :produce-models true)
(declare-const E Bool)
(declare-const Y Bool)
(declare-const L Bool)
(declare-const U Bool)
(declare-const T Bool)
(declare-const N Bool)
(declare-const C Bool)
(assert
  (and
    (or E Y L )
    (or L U L )
    (or T U N )
    (or C E L )
    (or (not E) (not Y) L )
    (or (not L) (not U) L )
    (or (not T) (not U) N )
    (or (not C) (not E) L )
  )
)
(check-sat)
(get-model)
(exit)
```

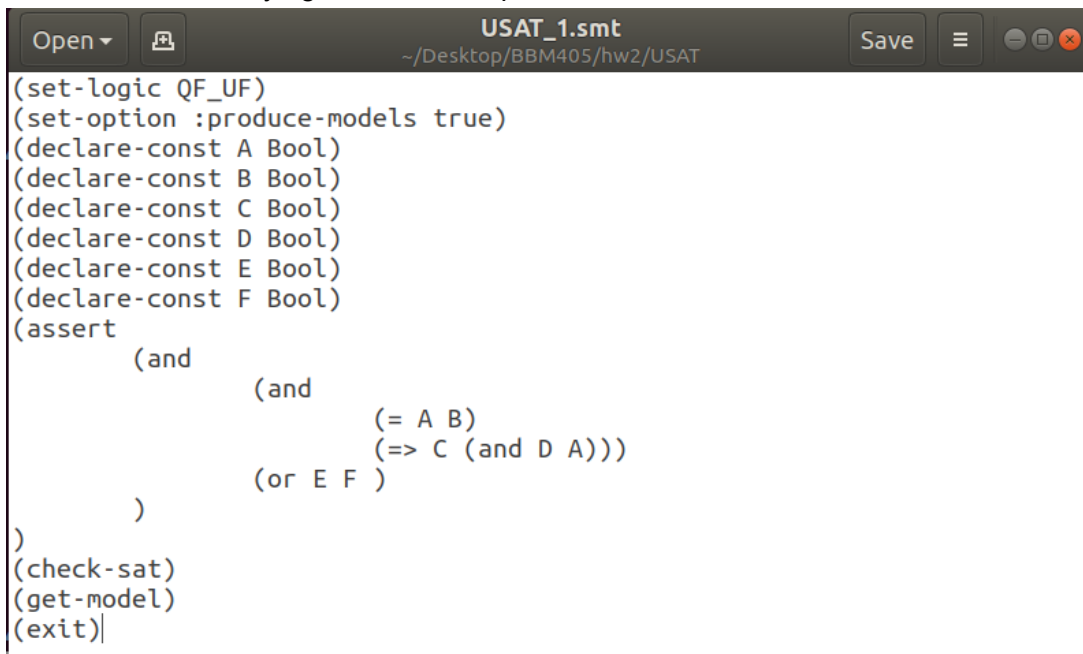
The model found by z3 is one of the satisfying values of the propositional formula. In the first line you can see that z3 checks for satisfiability and then finds it satisfiable. After that creates a model that has one instance of satisfying values. The values are also seen in the truth table.

```
eylul@kia:~/Desktop/BBM405/hw2$ z3 -smt2 3SAT_5.smt
sat
(
  (define-fun U () Bool
    true)
  (define-fun N () Bool
    false)
  (define-fun E () Bool
    false)
  (define-fun Y () Bool
    true)
  (define-fun T () Bool
    false)
  (define-fun L () Bool
    false)
  (define-fun C () Bool
    true)
)
```

3. **Part 2 : Write 5 unrestricted Propositional formulas. Convert them into CNF Then, convert them into 3-SAT Find the values satisfying the original and the final formula to show that they are equisatisfiable.**

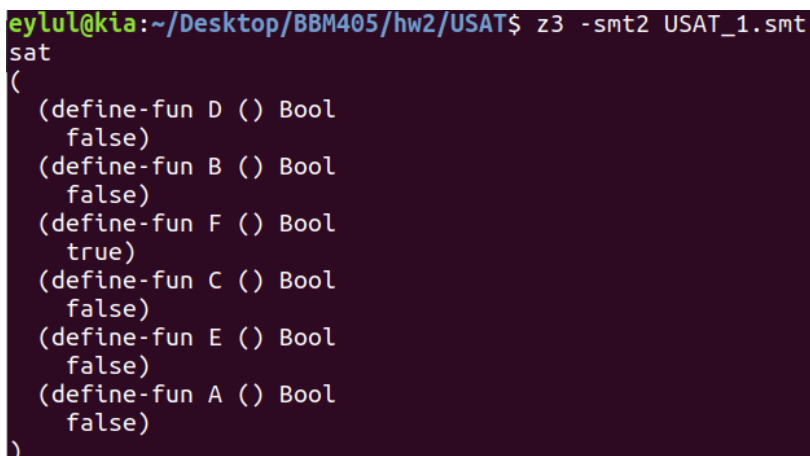
3.1.  $(a \Leftrightarrow b) \wedge (c \Rightarrow (d \wedge a)) \wedge (e \vee f)$

First we try our unrestricted formula on the z3 solver. And get the satisfying values of our problem. You can see below.



```
Open USAT_1.smt Save
~/Desktop/BBM405/hw2/USAT

(set-logic QF_UF)
(set-option :produce-models true)
(declare-const A Bool)
(declare-const B Bool)
(declare-const C Bool)
(declare-const D Bool)
(declare-const E Bool)
(declare-const F Bool)
(assert
  (and
    (and
      (= A B)
      (=> C (and D A)))
    (or E F )
  )
)
(check-sat)
(get-model)
(exit)|
```



```
eylul@kia:~/Desktop/BBM405/hw2/USAT$ z3 -smt2 USAT_1.smt
sat
(
  (define-fun D () Bool
    false)
  (define-fun B () Bool
    false)
  (define-fun F () Bool
    true)
  (define-fun C () Bool
    false)
  (define-fun E () Bool
    false)
  (define-fun A () Bool
    false)
)
```

After that we must convert our unrestricted formula to CNF, and then CNF to 3-SAT. Below you can find the operations on the unrestricted formula. I did all the operations one by one and at the end there is a 3-SAT version, equal to the firstly given propositional formula.

- 1)  $(a \Leftrightarrow b) \wedge (c \Rightarrow (d \wedge a)) \wedge (e \vee f)$
- 2)  $(a \Rightarrow b) \wedge (b \Rightarrow a) \wedge (c \Rightarrow (d \wedge a)) \wedge (e \vee f)$
- 3)  $(\neg a \vee b) \wedge (\neg b \vee a) \wedge (\neg c \vee (d \wedge a)) \wedge (e \vee f)$
- 4)  $(\neg a \vee b) \wedge (\neg b \vee a) \wedge (\neg c \vee d) \wedge (\neg c \vee a) \wedge (e \vee f)$   
 $= (\neg a \vee b)$   
 $\wedge (\neg b \vee a)$   
 $\wedge (\neg c \vee d)$   
 $\wedge (\neg c \vee a)$   
 $\wedge (e \vee f)$

```

USAT_1_converted.smt
~/Desktop/BBM405/hw2/USAT_CONVERTED
Open [icon] Save [icon] [icon] [icon]
(set-logic QF_UF)
(set-option :produce-models true)
(declare-const A Bool)
(declare-const B Bool)
(declare-const C Bool)
(declare-const D Bool)
(declare-const E Bool)
(declare-const F Bool)
(assert
  (and
    (or (not A) B )
    (or (not B) A )
    (or (not C) D )
    (or (not C) A ) |
    (or E F )
  )
)
(check-sat)
(get-model)
(exit)

```

```

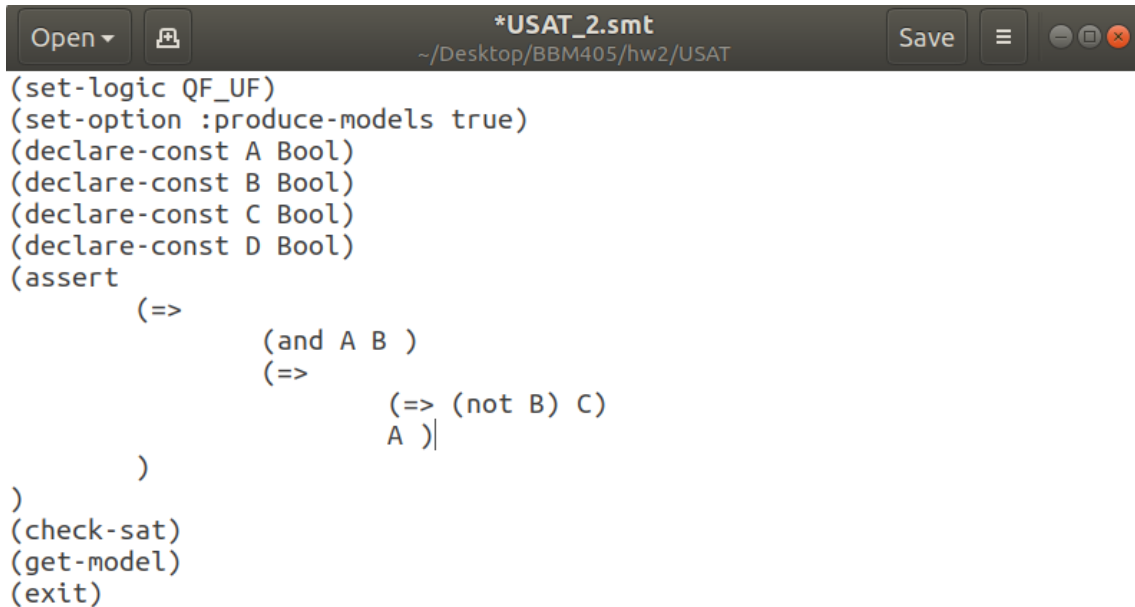
eylul@kita:~/Desktop/BBM405/hw2/USAT_CONVERTED$ z3 -smt2 USAT_1_converted.smt
sat
(
  (define-fun A () Bool
    false)
  (define-fun D () Bool
    false)
  (define-fun B () Bool
    false)
  (define-fun F () Bool
    false)
  (define-fun C () Bool
    false)
  (define-fun E () Bool
    true)
)

```

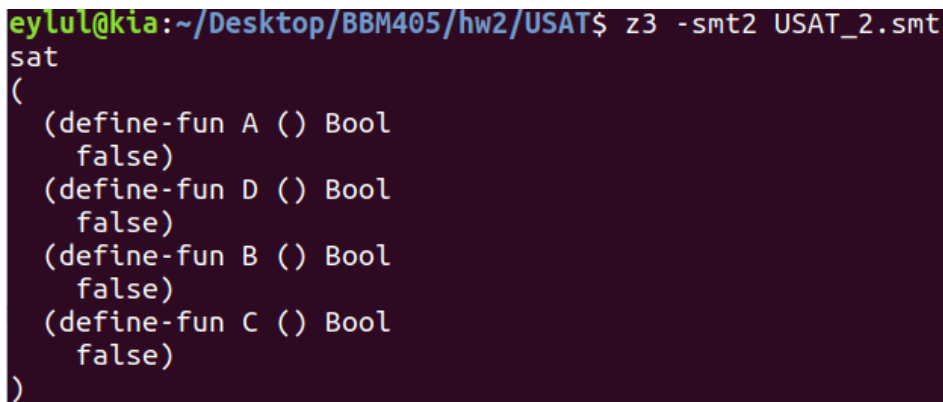
In logic, two formulas are equisatisfiable if the first formula is satisfiable whenever the second is and vice versa; in other words, either both formulae are satisfiable or both are not. We can see in both outputs that our two problems are satisfiable, so we can say that they are equisatisfiable.

### 3.2. $(a \vee b) \Rightarrow ((\neg b \Rightarrow c) \Rightarrow a)$

First we try our unrestricted formula on the z3 solver. And get the satisfying values of our problem. You can see below.



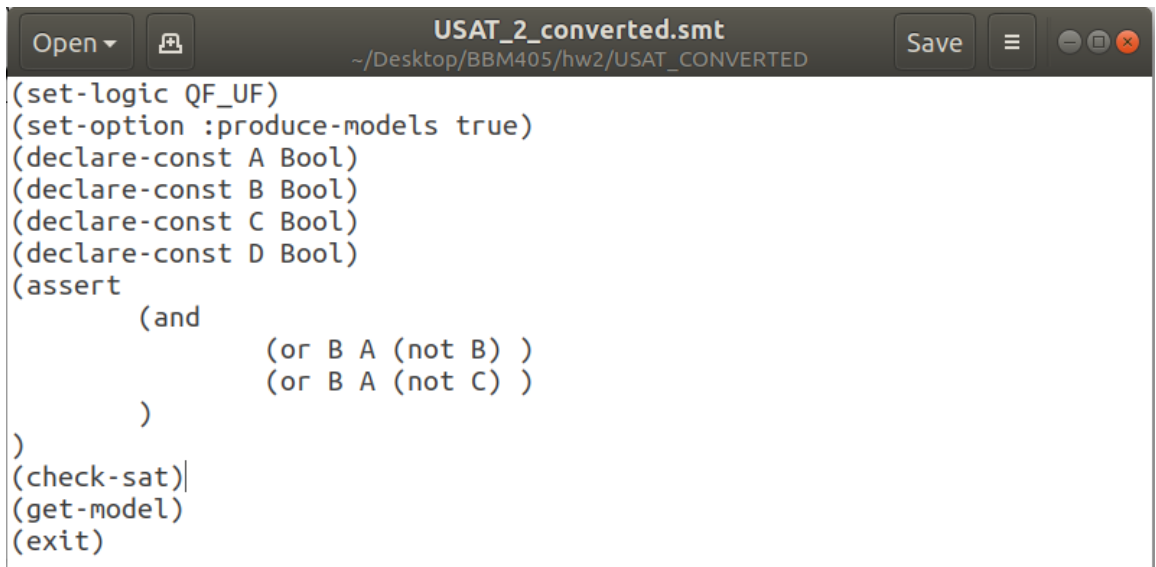
```
(set-logic QF_UF)
(set-option :produce-models true)
(declare-const A Bool)
(declare-const B Bool)
(declare-const C Bool)
(declare-const D Bool)
(assert
  (=>
    (and A B )
    (=>
      (=> (not B) C)
      A )
  )
)
(check-sat)
(get-model)
(exit)
```



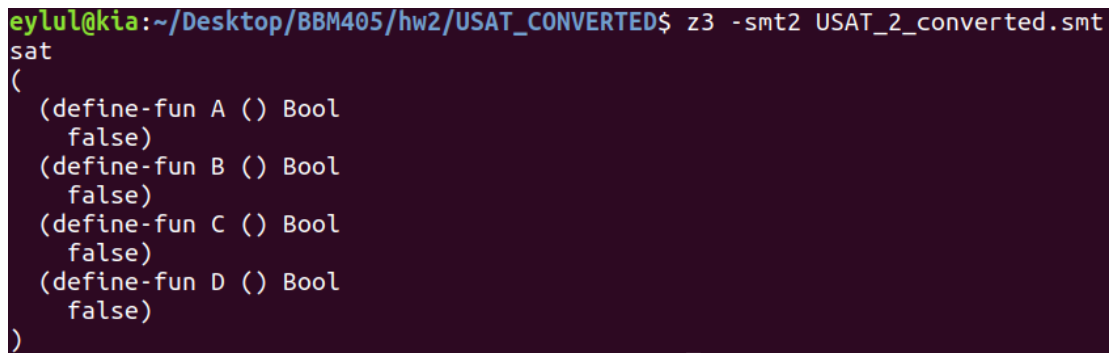
```
eylul@kia:~/Desktop/BBM405/hw2/USAT$ z3 -smt2 USAT_2.smt
sat
(
  (define-fun A () Bool
    false)
  (define-fun D () Bool
    false)
  (define-fun B () Bool
    false)
  (define-fun C () Bool
    false)
)
```

After that we must convert our unrestricted formula to CNF, and then CNF to 3-SAT. Below you can find the operations on the unrestricted formula. I did all the operations one by one and at the end there is a 3-SAT version, equal to the firstly given propositional formula.

- 1)  $(a \vee b) \Rightarrow ((\neg b \Rightarrow c) \Rightarrow a)$
- 2)  $\neg(a \vee b) \vee (\neg(b \vee c) \vee a)$
- 3)  $(\neg a \wedge b) \vee (\neg b \wedge \neg c) \vee a$
- 4)  $(\neg a \vee a) \wedge (b \vee a) \vee (\neg b \wedge \neg c)$
- 5)  $(1) \wedge (b \vee a) \vee (\neg b \wedge \neg c)$
- 6)  $(b \vee a) \vee (\neg b \wedge \neg c)$
- 7)  $(b \vee a \vee \neg b) \wedge (b \vee a \vee \neg c)$   
 $\quad = (b \vee a \vee \neg b)$   
 $\quad \wedge (b \vee a \vee \neg c)$



```
(set-logic QF_UF)
(set-option :produce-models true)
(declare-const A Bool)
(declare-const B Bool)
(declare-const C Bool)
(declare-const D Bool)
(assert
  (and
    (or B A (not B))
    (or B A (not C))
  )
)
(check-sat)
(get-model)
(exit)
```

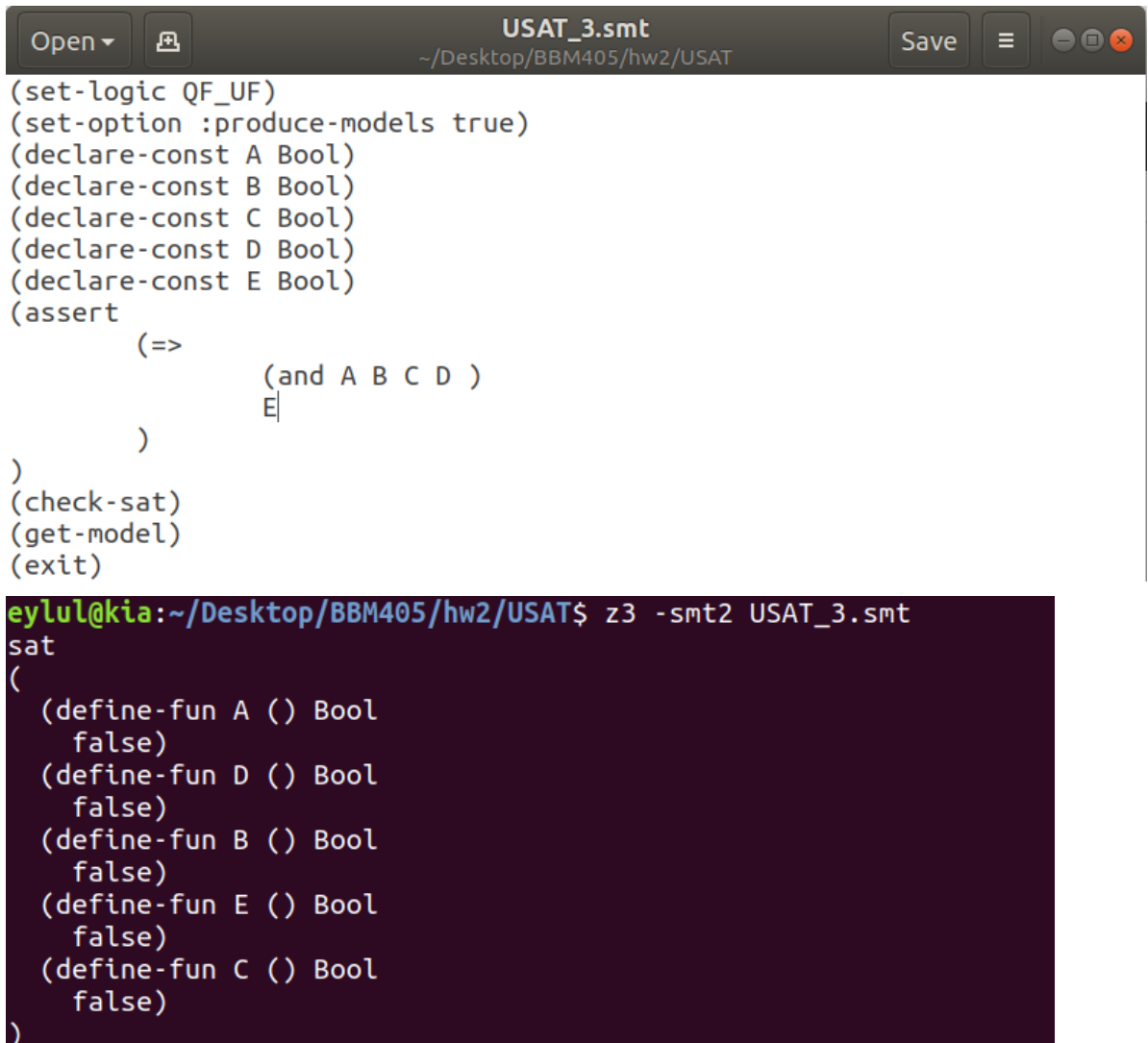


```
eylul@kia:~/Desktop/BBM405/hw2/USAT_CONVERTED$ z3 -smt2 USAT_2_converted.smt
sat
(
  (define-fun A () Bool
    false)
  (define-fun B () Bool
    false)
  (define-fun C () Bool
    false)
  (define-fun D () Bool
    false)
)
```

In logic, two formulas are equisatisfiable if the first formula is satisfiable whenever the second is and vice versa; in other words, either both formulae are satisfiable or both are not. We can see in both outputs that our two problems are satisfiable, so we can say that they are equisatisfiable.

### 3.3. $(a \wedge b \wedge c \wedge d) \Rightarrow e$

First we try our unrestricted formula on the z3 solver. And get the satisfying values of our problem. You can see below.



The screenshot shows a Z3 solver window titled "USAT\_3.smt" with the file path "~/Desktop/BBM405/hw2/USAT". The window contains the following SMT-LIB code:

```
(set-logic QF_UF)
(set-option :produce-models true)
(declare-const A Bool)
(declare-const B Bool)
(declare-const C Bool)
(declare-const D Bool)
(declare-const E Bool)
(assert
  (=>
    (and A B C D )
    E
  )
)
(check-sat)
(get-model)
(exit)
```

Below the window, a terminal window shows the command and output:

```
eylul@kia:~/Desktop/BBM405/hw2/USAT$ z3 -smt2 USAT_3.smt
sat
(
  (define-fun A () Bool
    false)
  (define-fun D () Bool
    false)
  (define-fun B () Bool
    false)
  (define-fun E () Bool
    false)
  (define-fun C () Bool
    false)
)
```

After that we must convert our unrestricted formula to CNF, and then CNF to 3-SAT. Below you can find the operations on the unrestricted formula. I did all the operations one by one and at the end there is a 3-SAT version, equal to the firstly given propositional formula.

- 1)  $(a \wedge b \wedge c \wedge d) \Rightarrow e$
- 2)  $\neg (a \wedge b \wedge c \wedge d) \vee e$
- 3)  $\neg a \vee \neg b \vee \neg c \vee \neg d \vee e$
- 4)  $(\neg a \vee b \vee x_1) \wedge (\neg x_1 \vee \neg c \vee x_2) \wedge (\neg x_2 \vee \neg d \vee x_3) \wedge (\neg x_3 \vee e)$   
     $= (\neg a \vee b \vee x_1)$   
     $\wedge (\neg x_1 \vee \neg c \vee x_2)$   
     $\wedge (\neg x_2 \vee \neg d \vee x_3)$   
     $\wedge (\neg x_3 \vee e)$

```
USAT_3_converted.smt
~/Desktop/BBM405/hw2/USAT_CONVERTED
Open Save

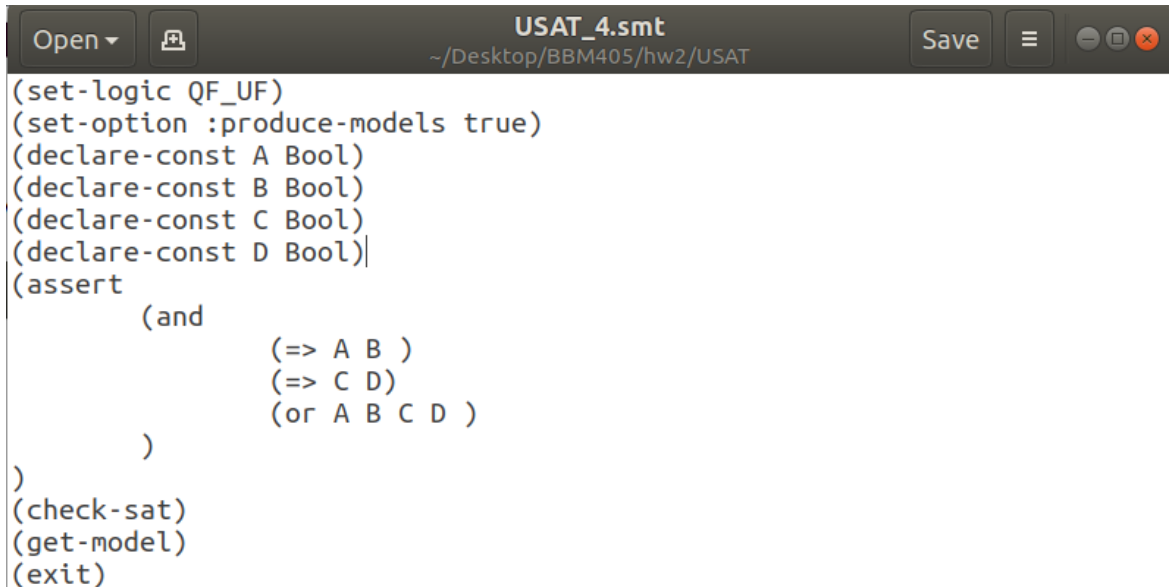
(set-logic QF_UF)
(set-option :produce-models true)
(declare-const A Bool)
(declare-const B Bool)
(declare-const C Bool)
(declare-const D Bool)
(declare-const E Bool)
(declare-const x1 Bool)
(declare-const x2 Bool)
(declare-const x3 Bool)
(assert
  (and
    (or (not A) B (not x1) )
    (or (not x1) (not C) x2 )
    (or (not x2) (not D) x3 )
    (or (not x3) E )
  ) |
)
(check-sat)
(get-model)
(exit)
```

```
eylul@kia:~/Desktop/BBM405/hw2/USAT_CONVERTED$ z3 -smt2 USAT_3_converted.smt
sat
(
  (define-fun A () Bool
    false)
  (define-fun D () Bool
    false)
  (define-fun E () Bool
    false)
  (define-fun x3 () Bool
    false)
  (define-fun x2 () Bool
    false)
  (define-fun B () Bool
    false)
  (define-fun x1 () Bool
    false)
  (define-fun C () Bool
    false)
)
```

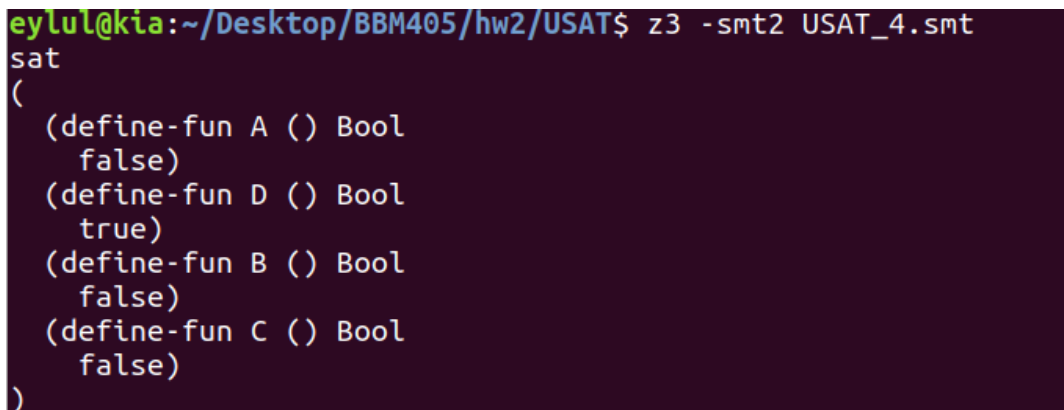
In logic, two formulas are equisatisfiable if the first formula is satisfiable whenever the second is and vice versa; in other words, either both formulae are satisfiable or both are not. We can see in both outputs that our two problems are satisfiable, so we can say that they are equisatisfiable.

### 3.4. $(a \Leftrightarrow b) \wedge (c \Rightarrow d) \wedge (a \vee b \vee c \vee d)$

First we try our unrestricted formula on the z3 solver. And get the satisfying values of our problem. You can see below.



```
(set-logic QF_UF)
(set-option :produce-models true)
(declare-const A Bool)
(declare-const B Bool)
(declare-const C Bool)
(declare-const D Bool)
(assert
  (and
    (=> A B )
    (=> C D)
    (or A B C D )
  )
)
(check-sat)
(get-model)
(exit)
```



```
eylul@kia:~/Desktop/BBM405/hw2/USAT$ z3 -smt2 USAT_4.smt
sat
(
  (define-fun A () Bool
    false)
  (define-fun D () Bool
    true)
  (define-fun B () Bool
    false)
  (define-fun C () Bool
    false)
)
```

After that we must convert our unrestricted formula to CNF, and then CNF to 3-SAT. Below you can find the operations on the unrestricted formula. I did all the operations one by one and at the end there is a 3-SAT version, equal to the firstly given propositional formula.

1.  $(a \Leftrightarrow b) \wedge (c \Rightarrow d) \wedge (a \vee b \vee c \vee d)$
2.  $(a \Rightarrow b) \wedge (b \Rightarrow a) \wedge (c \Rightarrow d) \wedge (a \vee b \vee c \vee d)$
3.  $(\neg a \vee b) \wedge (\neg b \wedge a) \wedge (\neg c \vee d) \wedge (a \vee b \vee c \vee d)$
4.  $(\neg a \vee b) \wedge (\neg b \wedge a) \wedge (\neg c \vee d) \wedge (a \vee b \vee x_1) \wedge (\neg x_1 \vee c \vee d)$   
     $= (\neg a \vee b)$   
     $\wedge (\neg b \wedge a)$   
     $\wedge (\neg c \vee d)$   
     $\wedge (a \vee b \vee x_1)$   
     $\wedge (\neg x_1 \vee c \vee d)$



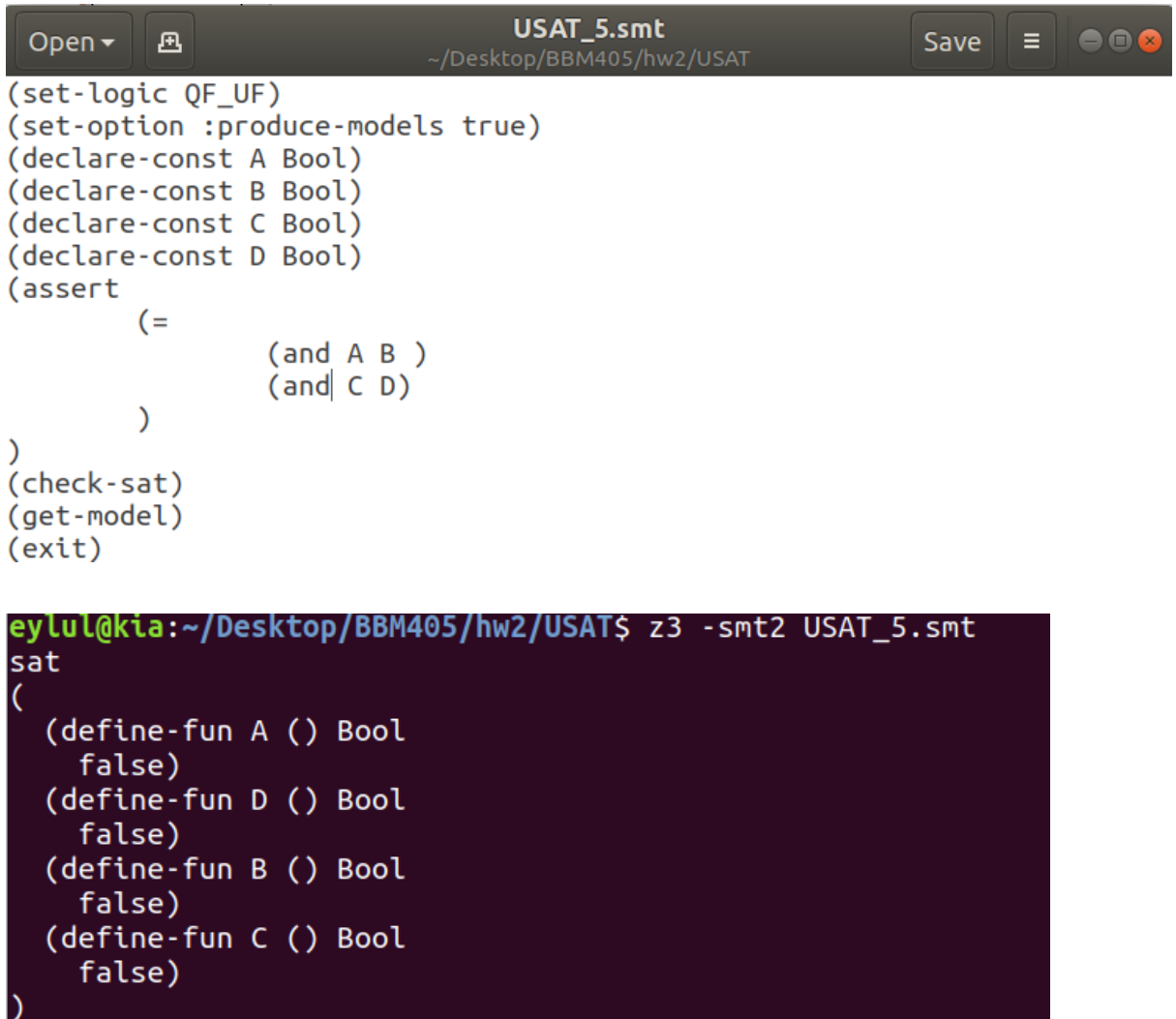
```
USAT_4_converted.smt
~/Desktop/BBM405/hw2/USAT_CONVERTED
Open Save
(set-logic QF_UF)
(set-option :produce-models true)
(declare-const A Bool)
(declare-const B Bool)
(declare-const C Bool)
(declare-const D Bool)
(declare-const x1 Bool)
(assert
  (and
    (or (not A) B )
    (or (not B) A )
    (or (not C) D )
    (or A B x1 )
    (or (not x1) C D)
  )
)
(check-sat)
(get-model)
(exit)
```

```
eylul@kia:~/Desktop/BBM405/hw2/USAT_CONVERTED$ z3 -smt2 USAT_4_converted.smt
sat
(
  (define-fun A () Bool
    false)
  (define-fun D () Bool
    true)
  (define-fun B () Bool
    false)
  (define-fun x1 () Bool
    true)
  (define-fun C () Bool
    false)
)
```

In logic, two formulas are equisatisfiable if the first formula is satisfiable whenever the second is and vice versa; in other words, either both formulae are satisfiable or both are not. We can see in both outputs that our two problems are satisfiable, so we can say that they are equisatisfiable.

### 3.5. $(a \wedge b) \Leftrightarrow (c \wedge d)$

First we try our unrestricted formula on the z3 solver. And get the satisfying values of our problem. You can see below.



The screenshot shows a Z3 solver window titled "USAT\_5.smt" with the file path "~/Desktop/BBM405/hw2/USAT". The window contains the following SMT-LIA code:

```
(set-logic QF_UF)
(set-option :produce-models true)
(declare-const A Bool)
(declare-const B Bool)
(declare-const C Bool)
(declare-const D Bool)
(assert
  (=
    (and A B)
    (and C D)
  )
)
(check-sat)
(get-model)
(exit)
```

Below the window, a terminal window shows the command and output:

```
eylul@kia:~/Desktop/BBM405/hw2/USAT$ z3 -smt2 USAT_5.smt
sat
(
  (define-fun A () Bool
    false)
  (define-fun D () Bool
    false)
  (define-fun B () Bool
    false)
  (define-fun C () Bool
    false)
)
```

After that we must convert our unrestricted formula to CNF, and then CNF to 3-SAT. Below you can find the operations on the unrestricted formula. I did all the operations one by one and at the end there is a 3-SAT version, equal to the firstly given propositional formula.

1.  $(a \wedge b) \Leftrightarrow (c \wedge d)$
  2.  $((a \wedge b) \Rightarrow (c \wedge d)) \wedge ((c \wedge d) \Rightarrow (a \wedge b))$
  3.  $(\neg(a \wedge b) \vee (c \wedge d)) \wedge (\neg(c \wedge d) \vee (a \wedge b))$
  4.  $((\neg a \vee \neg b) \vee (c \wedge d)) \wedge ((\neg c \vee \neg d) \vee (a \wedge b))$
  5.  $((c \vee \neg a \vee \neg b) \wedge (d \vee \neg a \vee \neg b)) \wedge ((a \vee \neg c \vee \neg d) \wedge (b \vee \neg c \vee \neg d))$
- $= (c \vee \neg a \vee \neg b)$   
 $\wedge (d \vee \neg a \vee \neg b)$   
 $\wedge (a \vee \neg c \vee \neg d)$   
 $\wedge (b \vee \neg c \vee \neg d)$

```
USAT_5_converted.smt
~/Desktop/BBM405/hw2/USAT_CONVERTED
Open Save
(set-logic QF_UF)
(set-option :produce-models true)
(declare-const A Bool)
(declare-const B Bool)
(declare-const C Bool)
(declare-const D Bool)
(assert
  (and
    (or C (not B) (not A) )
    (or D (not B) (not A) )
    (or A (not D) (not C) )
    (or B (not D) (not C) )
  )
)
(check-sat)
(get-model)
(exit)
```

```
eylul@kia:~/Desktop/BBM405/hw2/USAT_CONVERTED$ z3 -smt2 USAT_5_converted.smt
sat
(
  (define-fun A () Bool
    false)
  (define-fun D () Bool
    false)
  (define-fun B () Bool
    false)
  (define-fun C () Bool
    false)
)
```

In logic, two formulas are equisatisfiable if the first formula is satisfiable whenever the second is and vice versa; in other words, either both formulae are satisfiable or both are not. We can see in both outputs that our two problems are satisfiable, so we can say that they are equisatisfiable.

**4. Consider the algorithms discussed in the class for solving constraint satisfaction problems. You are now required to solve 3-SAT with generic CSP algorithms.**

For solving 3-SAT problems with generic CSP algorithms, I choose the backtracking algorithm. In the backtracking algorithm. Backtracking search is a depth-first search that chooses values for one variable at a time and backtracks when a variable has no values left to assign. Backtracking algorithm repeatedly chooses an unassigned variable, and then tries all values in the domain of that variable in turn, trying to find a solution.

For this purpose I use a library in Python that is named “constraint”. The Python constraint module offers solvers for Constraint Satisfaction Problems (CSPs) over finite domains. For implementing backtracking search in CSP problems we first define variables for the problem. And then for each variable define domains next to them. After completing the domain and variables then we can add constraints to the problem. Each constraint must be checked for final solution/s.

**4.1. Propositional Formula 1:**

$$(a \vee b) \wedge (b \vee c) \wedge (c \vee d) \wedge (d \vee a) \wedge (a \vee c) \\ \wedge (b \vee \neg c) \wedge (c \vee \neg d) \wedge (d \vee b)$$

```
92 problem = Problem(BacktrackingSolver())
93 problem.addVariables(["a", "b", "c", "d"], [True, False])
94 problem.addConstraint(lambda a, b, c, d: (a | b), ["a", "b", "c", "d"])
95 problem.addConstraint(lambda a, b, c, d: (b | c), ["a", "b", "c", "d"])
96 problem.addConstraint(lambda a, b, c, d: (c | d), ["a", "b", "c", "d"])
97 problem.addConstraint(lambda a, b, c, d: (d | a), ["a", "b", "c", "d"])
98 problem.addConstraint(lambda a, b, c, d: (a | c), ["a", "b", "c", "d"])
99 problem.addConstraint(lambda a, b, c, d: (b | (not c)), ["a", "b", "c", "d"])
100 problem.addConstraint(lambda a, b, c, d: (c | (not d)), ["a", "b", "c", "d"])
101 problem.addConstraint(lambda a, b, c, d: (d | b), ["a", "b", "c", "d"])
```

The output of the program is :

```
{'a': False, 'b': True, 'c': True, 'd': True}
{'a': True, 'b': True, 'c': True, 'd': True}
{'a': True, 'b': True, 'c': True, 'd': False}
```

The output of the program finds all the solutions that satisfy the constraints. The solutions found by our backtracking solver are matched with the solutions found in the first part. If we can look at the truth table in “2.1. Propositional Formula 1:” we can see that the values are the same.

#### 4.2. Propositional Formula 2 :

$(a \vee b \vee \neg c) \wedge (b \vee c \vee \neg d) \wedge (c \vee d \vee \neg a) \wedge (\neg a \vee \neg b \vee \neg d) \wedge (b \vee \neg c \vee \neg d)$

```
106 problem = Problem(BacktrackingSolver())
107 problem.addVariables(["a", "b", "c", "d"], [True, False])
108 problem.addConstraint(lambda a, b, c, d: (a | b | (not c)), ["a", "b", "c", "d"])
109 problem.addConstraint(lambda a, b, c, d: (b | c | (not d)), ["a", "b", "c", "d"])
110 problem.addConstraint(lambda a, b, c, d: (c | d | (not a)), ["a", "b", "c", "d"])
111 problem.addConstraint(lambda a, b, c, d: ((not a) | (not b) | (not d)), ["a", "b", "c", "d"])
112 problem.addConstraint(lambda a, b, c, d: (b | (not c) | (not d)), ["a", "b", "c", "d"])
```

The output of the program is :

```
{'a': False, 'b': False, 'c': False, 'd': False}
{'a': False, 'b': True, 'c': False, 'd': True}
{'a': False, 'b': True, 'c': False, 'd': False}
{'a': False, 'b': True, 'c': True, 'd': True}
{'a': False, 'b': True, 'c': True, 'd': False}
{'a': True, 'b': False, 'c': True, 'd': False}
{'a': True, 'b': True, 'c': True, 'd': False}
```

The output of the program finds all the solutions that satisfy the constraints. The solutions found by our backtracking solver are matched with the solutions found in the first part. If we can look at the truth table in “2.2. Propositional Formula 2:” we can see that the values are the same.

#### 4.3. Propositional Formula 3:

$(a \vee b \vee \neg c) \wedge (\neg b \vee d \vee e) \wedge (\neg a \vee b \vee c) \wedge (b \vee d \vee \neg e)$

```
116 problem = Problem(BacktrackingSolver())
117 problem.addVariables(["a", "b", "c", "d", "e"], [True, False])
118 problem.addConstraint(lambda a, b, c, d, e: (a | b | (not c)), ["a", "b", "c", "d", "e"])
119 problem.addConstraint(lambda a, b, c, d, e: ((not b) | d | e), ["a", "b", "c", "d", "e"])
120 problem.addConstraint(lambda a, b, c, d, e: ((not a) | b | c), ["a", "b", "c", "d", "e"])
121 problem.addConstraint(lambda a, b, c, d, e: (b | d | (not e)), ["a", "b", "c", "d", "e"])
```

The output of the program is :

```
{'a': False, 'b': False, 'c': False, 'd': False, 'e': False}
{'a': False, 'b': False, 'c': False, 'd': True, 'e': True}
{'a': False, 'b': False, 'c': False, 'd': True, 'e': False}
{'a': False, 'b': True, 'c': False, 'd': False, 'e': True}
{'a': False, 'b': True, 'c': False, 'd': True, 'e': False}
{'a': False, 'b': True, 'c': False, 'd': True, 'e': True}
{'a': False, 'b': True, 'c': True, 'd': False, 'e': True}
{'a': False, 'b': True, 'c': True, 'd': True, 'e': False}
{'a': False, 'b': True, 'c': True, 'd': True, 'e': True}
{'a': True, 'b': False, 'c': True, 'd': False, 'e': False}
{'a': True, 'b': False, 'c': True, 'd': True, 'e': True}
```

```
{'a': True, 'b': False, 'c': True, 'd': True, 'e': False}
{'a': True, 'b': True, 'c': False, 'd': False, 'e': True}
{'a': True, 'b': True, 'c': False, 'd': True, 'e': False}
{'a': True, 'b': True, 'c': False, 'd': True, 'e': True}
{'a': True, 'b': True, 'c': True, 'd': False, 'e': True}
{'a': True, 'b': True, 'c': True, 'd': True, 'e': False}
{'a': True, 'b': True, 'c': True, 'd': True, 'e': True}
```

The output of the program finds all the solutions that satisfy the constraints. The solutions found by our backtracking solver are matched with the solutions found in the first part. If we can look at the truth table in “2.3. Propositional Formula 3:” we can see that the values are the same.

4.4. Propositional Formula :

$$(\neg a \vee b) \wedge (\neg b \vee c) \wedge (\neg c \vee d) \wedge (\neg d \vee e) \wedge (a \vee \neg b) \wedge (b \vee \neg c) \wedge (c \vee \neg d) \wedge (d \vee \neg e)$$

```
127 problem = Problem(BacktrackingSolver())
128 problem.addVariables(["a", "b", "c", "d", "e"], [True, False])
129 problem.addConstraint(lambda a, b, c, d: ((not a) | b), ["a", "b", "c", "d"])
130 problem.addConstraint(lambda a, b, c, d: ((not b) | c), ["a", "b", "c", "d"])
131 problem.addConstraint(lambda a, b, c, d: ((not c) | d), ["a", "b", "c", "d"])
132 problem.addConstraint(lambda a, b, c, d: ((not d) | a), ["a", "b", "c", "d"])
133 problem.addConstraint(lambda a, b, c, d: (a | (not b)), ["a", "b", "c", "d"])
134 problem.addConstraint(lambda a, b, c, d: (b | (not c)), ["a", "b", "c", "d"])
135 problem.addConstraint(lambda a, b, c, d: (c | (not d)), ["a", "b", "c", "d"])
136 problem.addConstraint(lambda a, b, c, d, e: (d | (not e)), ["a", "b", "c", "d", "e"])
```

The output of the program is :

```
{'a': False, 'b': False, 'c': False, 'd': False, 'e': False}
{'a': True, 'b': True, 'c': True, 'd': True, 'e': True}
{'a': True, 'b': True, 'c': True, 'd': True, 'e': False}
```

The output of the program finds all the solutions that satisfy the constraints. The solutions found by our backtracking solver are matched with the solutions found in the first part. If we can look at the truth table in “2.4. Propositional Formula 4:” we can see that the values are the same.

#### 4.5. Propositional Formula :

$(e \vee y \vee l) \wedge (l \vee u \vee l) \wedge (t \vee u \vee n) \wedge (c \vee e \vee l) \wedge$   
 $(\neg e \vee \neg y \vee l) \wedge (\neg l \vee \neg u \vee l) \wedge (\neg t \vee \neg u \vee n) \wedge (\neg c \vee \neg e \vee l)$

```

140 problem = Problem(BacktrackingSolver())
141 problem.addVariables(["e", "y", "l", "u", "t", "n", "c"], [True, False])
142 problem.addConstraint(lambda e, y, l, u, t, n, c: (e | y | l), ["e", "y", "l", "u", "t", "n", "c"])
143 problem.addConstraint(lambda e, y, l, u, t, n, c: (l | u | l), ["e", "y", "l", "u", "t", "n", "c"])
144 problem.addConstraint(lambda e, y, l, u, t, n, c: (t | u | n), ["e", "y", "l", "u", "t", "n", "c"])
145 problem.addConstraint(lambda e, y, l, u, t, n, c: (c | e | l), ["e", "y", "l", "u", "t", "n", "c"])
146 problem.addConstraint(lambda e, y, l, u, t, n, c: ((not e) | (not y) | l), ["e", "y", "l", "u", "t", "n", "c"])
147 problem.addConstraint(lambda e, y, l, u, t, n, c: ((not l) | (not u) | l), ["e", "y", "l", "u", "t", "n", "c"])
148 problem.addConstraint(lambda e, y, l, u, t, n, c: ((not t) | (not u) | n), ["e", "y", "l", "u", "t", "n", "c"])
149 problem.addConstraint(lambda e, y, l, u, t, n, c: ((not c) | (not e) | l), ["e", "y", "l", "u", "t", "n", "c"])

```

The output of the program is :

```

{'c': False, 'e': False, 'l': True, 'n': False, 't': False, 'u': True, 'y': True}
{'c': False, 'e': False, 'l': True, 'n': False, 't': False, 'u': True, 'y': False}
{'c': False, 'e': False, 'l': True, 'n': False, 't': True, 'u': False, 'y': True}
{'c': False, 'e': False, 'l': True, 'n': False, 't': True, 'u': False, 'y': False}
{'c': False, 'e': False, 'l': True, 'n': True, 't': False, 'u': False, 'y': True}
{'c': False, 'e': False, 'l': True, 'n': True, 't': False, 'u': False, 'y': False}
{'c': False, 'e': False, 'l': True, 'n': True, 't': False, 'u': True, 'y': True}
{'c': False, 'e': False, 'l': True, 'n': True, 't': False, 'u': True, 'y': False}
{'c': False, 'e': False, 'l': True, 'n': True, 't': True, 'u': False, 'y': True}
{'c': False, 'e': False, 'l': True, 'n': True, 't': True, 'u': False, 'y': False}
{'c': False, 'e': False, 'l': True, 'n': True, 't': True, 'u': True, 'y': True}
{'c': False, 'e': False, 'l': True, 'n': True, 't': True, 'u': True, 'y': False}
{'c': False, 'e': True, 'l': False, 'n': False, 't': False, 'u': True, 'y': False}
{'c': False, 'e': True, 'l': False, 'n': True, 't': False, 'u': True, 'y': False}
{'c': False, 'e': True, 'l': False, 'n': True, 't': True, 'u': True, 'y': False}
{'c': False, 'e': True, 'l': True, 'n': False, 't': False, 'u': True, 'y': True}
{'c': False, 'e': True, 'l': True, 'n': False, 't': False, 'u': True, 'y': False}
{'c': False, 'e': True, 'l': True, 'n': False, 't': True, 'u': False, 'y': True}
{'c': False, 'e': True, 'l': True, 'n': False, 't': True, 'u': False, 'y': False}
{'c': False, 'e': True, 'l': True, 'n': True, 't': False, 'u': False, 'y': True}
{'c': False, 'e': True, 'l': True, 'n': True, 't': False, 'u': False, 'y': False}
{'c': False, 'e': True, 'l': True, 'n': True, 't': False, 'u': True, 'y': True}
{'c': False, 'e': True, 'l': True, 'n': True, 't': False, 'u': True, 'y': False}
{'c': False, 'e': True, 'l': True, 'n': True, 't': True, 'u': False, 'y': True}
{'c': False, 'e': True, 'l': True, 'n': True, 't': True, 'u': False, 'y': False}
{'c': False, 'e': True, 'l': True, 'n': True, 't': True, 'u': True, 'y': True}
{'c': False, 'e': True, 'l': True, 'n': True, 't': True, 'u': True, 'y': False}
{'c': True, 'e': False, 'l': False, 'n': False, 't': False, 'u': True, 'y': True}
{'c': True, 'e': False, 'l': False, 'n': True, 't': False, 'u': True, 'y': True}
{'c': True, 'e': False, 'l': False, 'n': True, 't': True, 'u': True, 'y': True}
{'c': True, 'e': False, 'l': True, 'n': False, 't': False, 'u': True, 'y': False}
{'c': True, 'e': False, 'l': True, 'n': False, 't': False, 'u': True, 'y': True}
{'c': True, 'e': False, 'l': True, 'n': False, 't': True, 'u': False, 'y': False}
{'c': True, 'e': False, 'l': True, 'n': False, 't': True, 'u': False, 'y': True}
{'c': True, 'e': False, 'l': True, 'n': True, 't': False, 'u': False, 'y': False}
{'c': True, 'e': False, 'l': True, 'n': True, 't': False, 'u': False, 'y': True}
{'c': True, 'e': False, 'l': True, 'n': True, 't': False, 'u': True, 'y': False}
{'c': True, 'e': False, 'l': True, 'n': True, 't': False, 'u': True, 'y': True}
{'c': True, 'e': False, 'l': True, 'n': True, 't': True, 'u': False, 'y': False}
{'c': True, 'e': False, 'l': True, 'n': True, 't': True, 'u': False, 'y': True}

```

```
{'c': True, 'e': False, 'l': True, 'n': True, 't': True, 'u': True, 'y': False}
{'c': True, 'e': False, 'l': True, 'n': True, 't': True, 'u': True, 'y': True}
{'c': True, 'e': True, 'l': True, 'n': False, 't': False, 'u': True, 'y': False}
{'c': True, 'e': True, 'l': True, 'n': False, 't': False, 'u': True, 'y': True}
{'c': True, 'e': True, 'l': True, 'n': False, 't': True, 'u': False, 'y': False}
{'c': True, 'e': True, 'l': True, 'n': False, 't': True, 'u': False, 'y': True}
{'c': True, 'e': True, 'l': True, 'n': True, 't': False, 'u': False, 'y': False}
{'c': True, 'e': True, 'l': True, 'n': True, 't': False, 'u': False, 'y': True}
{'c': True, 'e': True, 'l': True, 'n': True, 't': False, 'u': True, 'y': False}
{'c': True, 'e': True, 'l': True, 'n': True, 't': False, 'u': True, 'y': True}
{'c': True, 'e': True, 'l': True, 'n': True, 't': True, 'u': False, 'y': False}
{'c': True, 'e': True, 'l': True, 'n': True, 't': True, 'u': False, 'y': True}
{'c': True, 'e': True, 'l': True, 'n': True, 't': True, 'u': True, 'y': False}
{'c': True, 'e': True, 'l': True, 'n': True, 't': True, 'u': True, 'y': True}
```

The output of the program finds all the solutions that satisfy the constraints. The solutions found by our backtracking solver are matched with the solutions found in the first part. If we can look at the truth table in “2.5. Propositional Formula 5:” we can see that the values are the same



**5. Provide a detailed and informative discussion with your own thoughts and words.**

Firstly I wanted to start with what we did in this homework. In this homework we were required to solve satisfiability problems using different approaches. SAT (Satisfiability Problem) is the problem of determining whether there exists satisfying values to the given formula or sentence in propositional logic. We give our variables in this homework as boolean variables and constraints are in boolean logic. In SAT, variables are consistently assigned to the values True or False and check whether the interpretation satisfies the given constraints.

In the first approach, we use the Z3 Theorem Prover. Z3 is an efficient SMT solver which integrates DPLL-based SAT solver. DPLL (Davis Putnam Logemann Loveland) algorithm is a complete, backtracking-based search algorithm for deciding the satisfiability of propositional logic formulas in CNF. As I research how z3 works, I understand that z3 is doing assignments to the values one by one and then backtrack when faced with contradictions. This is very similar to the backtracking algorithm we learned in CSP I assume.

In the second approach, I use one of the CSP algorithms. The algorithm I choose is the backtracking algorithm. This is a very simple algorithm used for many varieties of problems. It may take a lot of time because in the backtracking algorithm we must reach the deadend have a conflict to backtrack. The more complex the problem, the slower the backtracking algorithm becomes. In Constraint Satisfaction Problems we have variables, domain and constraints. In each step the backtracking algorithm makes an assignment to a variable and checks for the constraints whether they are satisfied or not. Boolean CSPs can be NP-complete.

Both approaches work in a finite set of variables and constraints. A CSP is defined by a finite set of variables that take values from finite domains and by a finite set of constraints that restrict the values that the variables can simultaneously take.

In my homework, when I work with two of the approaches(z3 and backtracking csp), both work really well. Z3 gives satisfying values in seconds and also the backtracking algorithm of CSP gives satisfying values as output in seconds. One does not overcome the other. I think that is because of two things. One of them is that our SAT problems are really simple for a machine to solve, and the other thing is I think both approaches do the similar things behind. They both use similar approaches, mostly like backtracking. Maybe the Z3 theorem prover has some specialities that increase its performance but in our cases those improvements cannot be seen due to simplicity of problems.

**6. Bonus Task : Find another CSP problem, solve it. Then, formulate it as SAT and provide the solutions with Z3.**

**Sudoku:**

A simpler version of sudoku which only has 4 values and squares of size 2x2. The rules are the same. On the same square there are distinct numbers and on the rows and columns there are distinct numbers.

A1	A2	A3	A4
B1	B2	B3	B4
C1	C2	C3	C4
D1	D2	D3	D4

First I need to define the problem as CSP. Below you can see the code of the CSP problem solved with the backtracking algorithm.

```
from constraint import Problem, BacktrackingSolver

problem = Problem(BacktrackingSolver())
problem.addVariables(["a1", "a2", "a3", "a4",
                    "b1", "b2", "b3", "b4",
                    "c1", "c2", "c3", "c4",
                    "d1", "d2", "d3", "d4"],
                    [1, 2, 3, 4])

distinct_numbers = lambda a1, a2, b1, b2: (
    (a1 != a2) and (a1 != b1) and (a1 != b2) and (a2 != b1) and (a2 != b2) and (b1 != b2))

problem.addConstraint(distinct_numbers, ["a1", "a2", "b1", "b2"])
problem.addConstraint(distinct_numbers, ["a3", "a4", "b3", "b4"])
problem.addConstraint(distinct_numbers, ["c1", "c2", "d1", "d2"])
problem.addConstraint(distinct_numbers, ["c3", "c4", "d3", "d4"])

problem.addConstraint(distinct_numbers, ["a1", "a2", "a3", "a4"])
problem.addConstraint(distinct_numbers, ["b1", "b2", "b3", "b4"])
problem.addConstraint(distinct_numbers, ["c1", "c2", "c3", "c4"])
problem.addConstraint(distinct_numbers, ["d1", "d2", "d3", "d4"])

problem.addConstraint(distinct_numbers, ["a1", "b1", "c1", "d1"])
problem.addConstraint(distinct_numbers, ["a2", "b2", "c2", "d2"])
problem.addConstraint(distinct_numbers, ["a3", "b3", "c3", "d3"])
problem.addConstraint(distinct_numbers, ["a4", "b4", "c4", "d4"])
```

All squares are defined as separate variables with domain [1,2,3,4]. And then I add constraints to those squares. Each constraint defines either rule of each box, row or column.

A solution to the given python code (output of the program) :

```
a1 : 4      a2 : 3      a3 : 2      a4 : 1
b1 : 2      b2 : 1      b3 : 4      b4 : 3
c1 : 3      d1 : 1      c2 : 4      d2 : 2
c3 : 1      c4 : 2      d3 : 3      d4 : 4
Process finished with exit code 0
```

This solution satisfies all the constraints. Each box has distinct numbers. Each row has distinct numbers and also each row has distinct numbers.

Now we must implement this CSP problem on the Z3 Theorem prover. I converted this CSP problem to the 3-SAT and solve with Z3.

```

Open  sudoku.smt  Save  ~/Desktop/BBM405/hw2/Z3/sudoku
(set-logic QF_LIA)
(set-option :produce-models true)
(declare-const A1 Int)
(declare-const A2 Int)
(declare-const A3 Int)
(declare-const A4 Int)
(declare-const B1 Int)
(declare-const B2 Int)
(declare-const B3 Int)
(declare-const B4 Int)
(declare-const C1 Int)
(declare-const C2 Int)
(declare-const C3 Int)
(declare-const C4 Int)
(declare-const D1 Int)
(declare-const D2 Int)
(declare-const D3 Int)
(declare-const D4 Int)

(assert ( and (<= A1 4) (>= A1 1) (<= A2 4) (>= A2 1) ))
(assert ( and (<= A3 4) (>= A3 1) (<= A4 4) (>= A4 1) ))
(assert ( and (<= B1 4) (>= B1 1) (<= B2 4) (>= B2 1) ))
(assert ( and (<= B3 4) (>= B3 1) (<= B4 4) (>= B4 1) ))
(assert ( and (<= C1 4) (>= C1 1) (<= C2 4) (>= C2 1) ))
(assert ( and (<= C3 4) (>= C3 1) (<= C4 4) (>= C4 1) ))
(assert ( and (<= D1 4) (>= D1 1) (<= D2 4) (>= D2 1) ))
(assert ( and (<= D3 4) (>= D3 1) (<= D4 4) (>= D4 1) ))

(assert (distinct A1 A2 B1 B2))
(assert (distinct A3 A4 B3 B4))
(assert (distinct C1 C2 D1 D2))
(assert (distinct C3 C4 D3 D4))

(assert (distinct A1 A2 A3 A4))
(assert (distinct B1 B2 B3 B4))
(assert (distinct C1 C2 C3 C4))
(assert (distinct D1 D2 D3 D4))

(assert (distinct A1 B1 C1 D1))
(assert (distinct A2 B2 C2 D2))
(assert (distinct A3 B3 C3 D3))
(assert (distinct A4 B4 C4 D4))

```

Same steps are taken on the z3 theorem prover. And below there is an output.

```
eylul@kia:~/Desktop/BBM405/hw2/Z3/sudoku$ z3 -smt2 sudoku.smt
sat
(
  (define-fun A1 () Int
    4)
  (define-fun D1 () Int
    3)
  (define-fun C4 () Int
    3)
  (define-fun D4 () Int
    4)
  (define-fun A2 () Int
    2)
  (define-fun C1 () Int
    2)
  (define-fun D2 () Int
    1)
  (define-fun B1 () Int
    1)
  (define-fun D3 () Int
    2)
  (define-fun C2 () Int
    4)
  (define-fun B4 () Int
    2)
  (define-fun A4 () Int
    1)
  (define-fun B3 () Int
    4)
  (define-fun B2 () Int
    3)
  (define-fun C3 () Int
    1)
  (define-fun A3 () Int
    3)
)
```

## REFERENCES :

- [1] <https://pypi.org/project/python-constraint/>
- [2] <https://web.stanford.edu/class/cs103/tools/truth-table-tool/>
- [3] <https://www.cnblogs.com/RDaneelOlivaw/p/8072603.html>
- [4] <https://ericpony.github.io/z3py-tutorial/guide-examples.htm>
- [5] <https://en.wikipedia.org/wiki/Equisatisfiability>
- [6] <https://ericpony.github.io/z3py-tutorial/guide-examples.htm>
- [7] <https://www.microsoft.com/en-us/research/blog/the-inner-magic-behind-the-z3-theorem-prover/>