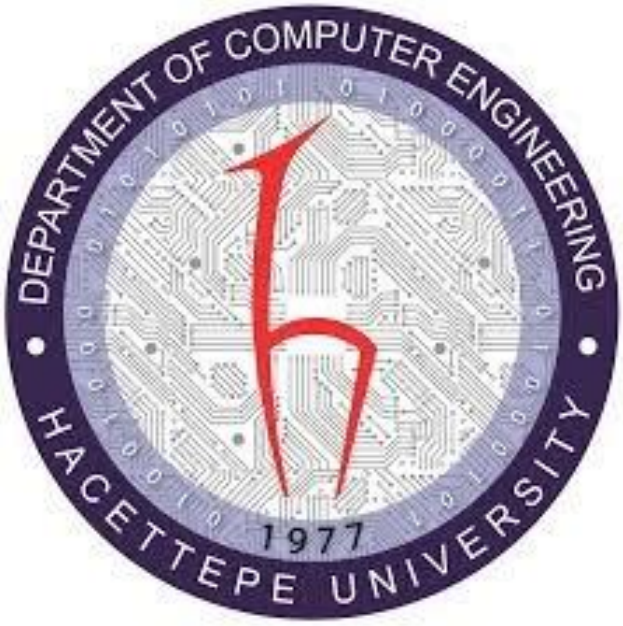
BBM405
Fundamentals of Artificial Intelligence
HW2



Name – Surname: Ozar Ömer Uncu

Student Id: 21727815

Introduction:

Aim of this homework is solve constrain satisfaction problems by using some tools. These problems are logical problems states number of constraints or limitations. They can solve by constraint satisfaction methods. We will use python libraries and Z3 tool for solve these problems by computer and see what our output at these situations.

Part 1:

1. $(X \vee Z \vee (X \vee \neg Y)) \wedge (\neg X \vee \neg Y \vee (X \vee \neg Z)) \wedge (X \vee (Y \vee (Z \vee (X \vee \neg Y))))$

```
(set-logic QF_UF)
(set-option :produce-models true)
(declare-const X Bool)
(declare-const Y Bool)
(declare-const Z Bool)
(assert
    (and
        (or
            (or X Z)
            (or X (not Y))
        )
        (or
            (or (not X) (not Y))
            (or X (not Z))
        )
        (or X (or Y (or Z (or X (not Y)))))
    )
)
(check-sat)
(get-model)
(exit)
```

```
sat
(model
  (define-fun Z () Bool
    false)
  (define-fun X () Bool
    false)
  (define-fun Y () Bool
    false)
)
```

2. $(X \vee (\neg X \vee Z)) \wedge (X \vee \neg Y \vee Z \vee (\neg X \vee Y))$

```
(set-logic QF_UF)
(set-option :produce-models true)
(declare-const X Bool)
(declare-const Y Bool)
(declare-const Z Bool)
(assert
    (and
        (or X (or (not X) Z))
        (or X (not Y) Z (or (not X) Y))
    )
)
(check-sat)
(get-model)
(exit)
```

```
sat
(model
  (define-fun Z () Bool
    false)
  (define-fun X () Bool
    false)
  (define-fun Y () Bool
    false)
)
```

3. (X v Y v W v ¬Z) ∧ (X v ¬Y v W v ¬Z)

```
(set-logic QF_UF)
(set-option :produce-models true)
(declare-const X Bool)
(declare-const Y Bool)
(declare-const Z Bool)
(declare-const W Bool)

(assert
    (and
        (or X Y W (not Z))
        (or X (not Y) W (not Z))
    )
)
(check-sat)
(get-model)
(exit)
```

```
sat
(model
  (define-fun Z () Bool
    false)
  (define-fun X () Bool
    false)
  (define-fun W () Bool
    false)
  (define-fun Y () Bool
    false)
)
```

4. (X v (W v Z)) ∧ (¬X v (Y v Z)) ∧ (X v (Y v W)) ∧ (Z v (Y v W))

```
(set-logic QF_UF)
(set-option :produce-models true)
(declare-const X Bool)
(declare-const Y Bool)
(declare-const Z Bool)
(declare-const W Bool)

(assert
    (and
        (or X (or W Z))
        (or X (or Y Z))
        (or X (or Y W))
        (or Z (or Y W))
    )
)
(check-sat)
(get-model)
(exit)
```

```
(model
  (define-fun Z () Bool
    false)
  (define-fun X () Bool
    false)
  (define-fun W () Bool
    true)
  (define-fun Y () Bool
    true)
)
```

5. (X v Y) ∧ (X v Z) ∧ (Y v Z) ∧ (X v (Y v ¬W)) ∧ (X v ¬W) ∧ (Z v (Y v W)) ∧ (Y v W) ∧ (Z v W)

```
(set-logic QF_UF)
(set-option :produce-models true)
(declare-const X Bool)
(declare-const Y Bool)
(declare-const Z Bool)
(declare-const W Bool)

(assert
    (and
        (or X Y)
        (or X Z)
        (or Y Z)
        (or X (or Y (not W)))
        (or X (not W))
        (or Z (or Y W))
        (or Y W)
        (or Z W)
    )
)
(check-sat)
(get-model)
(exit)
```

```
(model
  (define-fun Z () Bool
    true)
  (define-fun X () Bool
    false)
  (define-fun W () Bool
    false)
  (define-fun Y () Bool
    true)
)
```

For the part 1 you can see propositional formulas in CNF at above.

We use z3 -smt2 3SAT_*.smt code for the run programs.

Part 2:

! Some situations be simplified like:
- X v ¬X
- (X v Y) ∧ (X v Y)

1. (X => (Y v Z)) ∧ (Y => (¬X v Z)) ∧ (Z => (X v Y v Z))

```
(set-logic QF_UF)
(set-option :produce-models true)
(declare-const X Bool)
(declare-const Y Bool)
(declare-const Z Bool)

(assert
    (and
        (=> X (or Y Z))
        (=> Y (or (not X) Z))
        (=> Z (or X (or Y Z)))
    )
)
(check-sat)
(get-model)
(exit)
```

```
(model
  (define-fun Z () Bool
    false)
  (define-fun X () Bool
    false)
  (define-fun Y () Bool
    false)
)
```

Converted:

(¬X v (Y v Z)) ∧(¬Y v (¬X v Z)) ∧ (¬Z v (X v Y v Z))

```
(set-logic QF_UF)
(set-option :produce-models true)
(declare-const X Bool)
(declare-const Y Bool)
(declare-const Z Bool)

(assert
    (and
        (or (not X)
            (or Y Z))
        (or (not Y)
            (or (not X) Z))
        (or (not Z)
            (or X Y Z))
    )
)
(check-sat)
(get-model)
(exit)
```

```
smt
sat
(model
  (define-fun Z () Bool
    false)
  (define-fun X () Bool
    false)
  (define-fun Y () Bool
    false)
)
```

2. ((X v Z) ⇔ (Y v ¬X)) ∧ ((X v Y v Z) ⇔ (X v ¬Y v Z))

```
(set-logic QF_UF)
(set-option :produce-models true)
(declare-const X Bool)
(declare-const Y Bool)
(declare-const Z Bool)

(assert
    (and
        (=
            (or X Z)
            (or Y (not X))
        )
        (=
            (or X (or Y Z))
            (or X (or (not Y) Z))
        )
    )
)
(check-sat)
(get-model)
(exit)
```

```
(model
  (define-fun Z () Bool
    true)
  (define-fun X () Bool
    false)
  (define-fun Y () Bool
    false)
)
```

Converted:

(¬X v Y) ∧ (Z v Y v ¬X) ∧ (¬Y v X v Z) ∧ (X v Z) ∧ (¬Y v Z) ∧ (X v Y) ∧ (Y v Z) ∧ (Y v X v Z)

```
(set-logic QF_UF)
(set-option :produce-models true)
(declare-const X Bool)
(declare-const Y Bool)
(declare-const Z Bool)

(assert
    (and
        (or (not X) Y)
        (or Z Y (not X))
        (or (not Y) X Z)
        (or Y Z)
        (or X Z)
        (or X Y)
        (or Y X Z)
    )
)
(check-sat)
(get-model)
(exit)
```

```
sat
(model
  (define-fun Z () Bool
    true)
  (define-fun X () Bool
    false)
  (define-fun Y () Bool
    true)
)
```

3. ((X => Z) ⇔ (Y => ¬X)) ∧ ((X v Y v ¬Z) ⇔ (X => (¬Y v Z)))

```
(set-logic QF_UF)
(set-option :produce-models true)
(declare-const X Bool)
(declare-const Y Bool)
(declare-const Z Bool)

(assert
    (and
        (=
            (=> X Z)
            (=> Y (not X))
        )
        (=
            (=> X (or Y Z))
            (=> X (or (not Y) Z))
        )
    )
)
(check-sat)
(get-model)
(exit)
```

```
(model
  (define-fun Z () Bool
    false)
  (define-fun X () Bool
    false)
  (define-fun Y () Bool
    false)
)
```

Converted:

(X v ¬Y) ∧ (¬Z v ¬Y v X) ∧ (Y v ¬X v Z) ∧ (¬X v Z) ∧ (¬Y v Z) ∧ (¬X v ¬Y v Z) ∧ (Y v Z) ∧ (¬X v Y v Z) ∧ (¬X v Y)

```
(set-logic QF_UF)
(set-option :produce-models true)
(declare-const X Bool)
(declare-const Y Bool)
(declare-const Z Bool)

(assert
    (and
        (or X (not Y))
        (or (not Z) (not Y) X)
        (or Y (not X) Z)
        (or (not X) Z)
        (or (not Y) Z)
        (or (not X) (not Y) Z)
        (or Y Z)
        (or (not X) Y Z)
        (or (not X) Y)
    )
)
(check-sat)
(get-model)
(exit)
```

```
sat
(model
  (define-fun Z () Bool
    true)
  (define-fun X () Bool
    false)
  (define-fun Y () Bool
    false)
)
```

4. ((X => W) ⇔ (Y => Z)) ∧ ((X => Q) ⇔ (Q => Y))

```
(set-logic QF_UF)
(set-option :produce-models true)
(declare-const X Bool)
(declare-const Y Bool)
(declare-const Z Bool)
(declare-const W Bool)
(declare-const Q Bool)

(assert
    (and
        (= (=> X W) (=> Y Z))
        (= (=> X Q) (=> Q (not Y)))
    )
)
(check-sat)
(get-model)
(exit)
```

```
  (define-fun Z () Bool
    false)
  (define-fun X () Bool
    false)
  (define-fun Q () Bool
    false)
  (define-fun W () Bool
    false)
  (define-fun Y () Bool
    false)
)
```

Converted:

(X v ¬Y v Z) ∧ (¬W v ¬Y v Z) ∧ (Y v ¬X v W) ∧ (¬Z v ¬X v W) ∧ (¬Q v X v Y) ∧ (¬Q v Y) ∧ (¬X v Q) ∧ (¬Y v ¬X v Q)

```
(set-logic QF_UF)
(set-option :produce-models true)
(declare-const X Bool)
(declare-const Y Bool)
(declare-const Z Bool)
(declare-const W Bool)
(declare-const Q Bool)

(assert
    (and
        (or X (not Y) Z)
        (or (not Y) (not W) Z)
        (or Y (not X) W)
        (or (not Z) (not X) W)
        (or X (not Q) Y)
        (or (not Q) Y)
        (or (not X) Q)
        (or (not X) (not Y) Q)
    )
)
(check-sat)
(get-model)
(exit)
```

```
sat
(model
  (define-fun Z () Bool
    false)
  (define-fun X () Bool
    false)
  (define-fun Q () Bool
    false)
  (define-fun W () Bool
    false)
  (define-fun Y () Bool
    false)
)
```

5. ((X v Q v Z) => (Y => Z)) ∧ ((X=>Q) ⇔ (Y => Z)

```
(set-logic QF_UF)
(set-option :produce-models true)
(declare-const X Bool)
(declare-const Y Bool)
(declare-const Z Bool)
(declare-const W Bool)
(declare-const Q Bool)
(assert
    (and
        (=>
            (or X Q Z)
            (=> Y Z)
        )
        (=
            (=> X Q)
            (=> Y Z)
        )
    )
)
(check-sat)
(get-model)
(exit)
```

```
(model
  (define-fun Z () Bool
    false)
  (define-fun X () Bool
    false)
  (define-fun Q () Bool
    false)
  (define-fun Y () Bool
    false)
  (define-fun W () Bool
    false)
)
```

Converted:

(¬X v ¬Y v Z) ∧ (¬Q v ¬Y v Z) ∧ (¬Y) ∧ (X v ¬Y v Z) ∧ (¬Q v ¬Y v Z) ∧ (Y v ¬X v Q) ∧ (¬Z v ¬X v Q)

```
(set-logic QF_UF)
(set-option :produce-models true)
(declare-const X Bool)
(declare-const Y Bool)
(declare-const Z Bool)
(declare-const W Bool)
(declare-const Q Bool)
(assert
    (and
        (or (not X) (not Y) Z)
        (or (not Q) (not Y) Z)
        (or (not Y))
        (or X (not Y) Z)
        (or Y (not X) Q)
        (or (not Z) (not X) Q)
    )
)
(check-sat)
(get-model)
(exit)
```

```
sat
(model
  (define-fun Z () Bool
    false)
  (define-fun X () Bool
    false)
  (define-fun Q () Bool
    false)
  (define-fun Y () Bool
    false)
  (define-fun W () Bool
    false)
)
```

For the part 2 you can see unrestricted Propositional formulas in CNF at above. I really work to use all operators but sometimes I could not convert complex formulas. So, I choose easier formulas to show at homework.

We use z3 -smt2 USAT_*-org.smt for the original formulas code for the run programs.

We use z3 -smt2 USAT_*-converted.smt for the converted formulas code for the run programs.

Part 3:

For this part we used python PL. We use constraint library which contributes us problem solution methods. We give our elements to method to define them as True or False. For starting two problem we have 3 element X, Y and Z. Then I added W element to take our problems more complex. I print all output of these problems. These show us similarities with z3 outputs as well.

```
from constraint  import *

print("1st Problem")
problem = Problem(BacktrackingSolver())
problem.addVariables(["X","Y","Z"],[True,False])
problem.addConstraint(lambda X,Y,Z :  (X|Z|(X| (not Y))) & ((not X)|(not Y)|((not Z)|X)) & (X|(Y|(Z|(X|(not Y))))) , ["X","Y","Z"])
solution1 = problem.getSolution()
print(solution1)
print("---------------------")
print("2nd Problem")
problem1 = Problem(BacktrackingSolver())
problem1.addVariables(["X","Y","Z"],[True,False])
problem1.addConstraint(lambda X,Y,Z :  (X|((not X)|Z)) & (X|(not Y)|Z |((not X)|Y)) , ["X","Y","Z"])
solution2 = problem1.getSolution()
print(solution2)
print("---------------------")
print("3rd Problem")
problem2 = Problem(BacktrackingSolver())
problem2.addVariables(["X","Y","Z","W"],[True,False])
problem2.addConstraint(lambda X,Y,Z,W :  (X | Y | W | (not Z)) & (X | (not Y) | W | (not Z)) , ["X","Y","Z","W"])
solution3 = problem2.getSolution()
print(solution3)
print("---------------------")
print("4th Problem")
problem3 = Problem(BacktrackingSolver())
problem3.addVariables(["X","Y","Z","W"],[True,False])
problem3.addConstraint(lambda X,Y,Z,W :  (X | (W | Z)) & ((not X) | (Y | Z)) & (X | (Y | W)) & (Z | (Y | W)) , ["X","Y","Z","W"])
solution4 = problem3.getSolution()
print(solution4)
print("---------------------")
print("5th Problem")
problem4 = Problem(BacktrackingSolver())
problem4.addVariables(["X","Y","Z","W"],[True,False])
problem4.addConstraint(lambda X,Y,Z,W : (X | Y) & (X | Z) & (Y | Z) & (X | (Y | (not W))) & (X | (not W)) & (Z | (Y | W)) & (Y | W) & (Z | W)  , ["X","Y","Z","W"])
solution5 = problem4.getSolution()
print(solution5)
```

```
1st Problem
{'X': False, 'Y': False, 'Z': False}
---------------------
2nd Problem
{'X': False, 'Y': False, 'Z': False}
---------------------
3rd Problem
{'W': False, 'X': False, 'Y': False, 'Z': False}
---------------------
4th Problem
{'W': False, 'X': False, 'Y': True, 'Z': True}
---------------------
5th Problem
{'W': False, 'X': False, 'Y': True, 'Z': True}
```

Part 4:

At last, we work on different problems that we contribute to solve our machines. I see from the negotiation of elements can cause false outputs as well.

Z3 is a fast tool for predict about our problems like CSPs satisfiability or logical distributions of problems. From this homework we can see it easy to use for basic operands. Like or , and , implies , etc.. For our problem it works really fast I could not see any delay to see results of problems.

CSPs, we used python library called as “constraint”. This library provides us backtracking solver method for pass our problems to program. It also run as fast as z3. We did not observe delay at this function. I need to assign true or false values to problems method. It contributes us same output with the Z3.

I could not observe huge differences between two method we implemented. I can say there is nothing difference at difficulty of use or speed for our two implement methods.

References:

-https://en.wikipedia.org/wiki/Equisatisfiability

-https://web.cs.hacettepe.edu.tr/~pinar/courses/BBM405/index.html

-https://en.wikipedia.org/wiki/List_of_logic_symbols

-https://pypi.org/project/python-constraint/

-https://levelup.gitconnected.com/csp-algorithm-vs-backtracking-sudoku-304a242f96d0

-https://en.wikipedia.org/wiki/Boolean_satisfiability_problem

-https://en.wikipedia.org/wiki/Satisfiability_modulo_theories

-https://github.com/Z3Prover/z3

-https://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/index-seo.php/SMT____Z3-INSTALLATION

-https://rise4fun.com/z3/tutorial

-http://www.cs.ecu.edu/karl/6420/spr16/Notes/NPcomplete/3sat.html