CS301

2023-2024 Spring

FINAL REPORT

Group 004

::Group Member::

Murat Doğan

1. Problem Description

Name:	Maximum 2-Satisfiability
Input:	A set of m clauses $C_1,C_2,...,C_m$ over n Boolean valued variables X_n, where each clause depends on two distinct variables from X_n; a positive integer k with k<=m.
Question:	Is there an assignment of Boolean values to the variables X_n such that at least k distinct clauses take the value true under the assignment?

1.1 Overview

Maximum 2-Satisfiability problem contains Boolean valued variables and clauses. These clauses are formed by disjunction of two literals and each literal are one of X_n variable or negation of X_n variable. The aim of the problem is to find correct number of k clauses are satisfied with the assigned Boolean variable values. Also, k number must be largest possible number smaller than m for correctness of problem.

1.2 Decision Problem

Given set of m clauses with the k distinct clauses, is it feasible to make problem satisfied through correct placement of Boolean valued variables and true number of k clauses?

1.3 Optimization Problem

The aim is to maximize satisfied number of clauses with the given n number of Boolean valued variables.

1.4 Example Illustration

Example 1: Simple Case

$(x_1 \lor x_2)$

$(\neg x_1 \lor x_2)$

$(x_1 \lor \neg x_2)$

When x_1 = true and x_2 = true, all clauses are satisfied and $k = 3$.

Example 2: Satisfied k

$(x_1 \lor x_2)$

$(\neg x_1 \lor x_3)$

$(x_2 \lor \neg x_3)$

$(x_3 \lor x_1)$

When $k = 3$ and x_1 = true, x_2 = true and x_3 = false, 3 of these clauses are satisfied and $k = 3$. Even if 1 of them is not satisfied, problem is satisfied due to k value.

Example 3: Unsatisfied Problem

$(x_1 \lor x_2)$

$(\neg x_1 \lor x_2)$

$(x_1 \lor \neg x_2)$

$(\neg x_1 \lor \neg x_2)$

When $k = 3$, there are no possible Boolean value distribution for satisfied solution.

1.5 Real World Applications

1.5.1 Hardware Design

Max 2-SAT problem can be crucial for digital circuits in Hardware Design, it determines optimized version of digital circuits with the Boolean variable values.2-SAT
is easier to use than other SATs because it is not that complicated compared to others and gives optimistic results faster. Also, it can detect configuration errors with the unsatisfied clauses. Therefore, delays and issues are reduced when errors are resolved.

1.5.2 Software Verification

Max 2-SAT is helpful to verify software and countered problems. Test Cases can be checked with Boolean variables and generate the test cases' behavior. Also, it is beneficial for debugging, it can find all the unsatisfied clauses and sort them according to their importance level to gain time.

1.5.3 Cryptography and Security

Boolean variables are useful for cryptographic protocols, maximized true clauses make easier to predict number of correct keystream bits. In addition, it checks every possible condition for security protocols, with Maximum 2-SAT it can find different satisfied security constraints by best combined security rules.

1.6 Hardness of the Problem

Garey, Johnson and Stockmeyer proved that Maximum 2-Satisfiabilty is NP-complete by transforming 3-SAT instance F into Maximum 2-SAT instance F' by changing each clause of F.

Reduction Algorithm:

$C_i = (x_1 \vee x_2 \vee x_3)$

Reduced 2-CNF clauses:

(x_1),(x_2),(x_3),(C_i),$(\neg x_1 \lor \neg x_2)$,$(\neg x_2 \lor \neg x_3)$,$(\neg x_1 \lor \neg x_3)$,$(x_1 \lor \neg C_i)$,$(x_2 \lor \neg C_i)$,$(x_3 \lor \neg C_i)$

C_i is a new variable:

If it is satisfied C_i , then seven of ten clauses are satisfied.

If it is not satisfied C_i , then six of ten clauses are satisfied.

If F is satisfied, then 7 out of 10 clauses are satisfied in F'.
If F is not satisfied, then there is no assignment that satisfies 7 out of 10 clauses in F'.
Therefore, Maximum 2-Satisfiabilty is NP-complete.

2. Algorithm Description

2.1 Brute Force Algorithm

2.1.1 Overview

The correct/ exact/ brute force algorithm contains computing every possible value (True or False) for X_n variable. So, program generates 2^n feasible assignments. And this method assures that optimal solution will be found.

- Enumerate every possible variable combination.
- Check every satisfied clause and calculate total satisfied clause of every variable combination.
- Compare total satisfied clause number of that specific variable combination with Maximum total satisfied clause number from previously calculated combinations. If Maximum total satisfied clause number is undefined, assign that combination as Maximum total satisfied clause number.

- After checking all possible variable combination, return Maximum total satisfied clause number.

As a result, algorithm will find the optimal solution, but large n values can cause long computation times.

2.1.2 Pseudocode

```
*****************************************************************
Step 1:       Initialize MaxSatisfied = 0
Step 2:         For current_assignment in all_assignments
Step 2.1:         CurrentSatisfied = 0
Step 2.2:         For current_clause in all_clauses
Step 2.2. 1:             if current_clause is satisfied
Step 2.2. 2:                 CurrentSatisfied += 1
Step 2.3:         if CurrentSatisfied > MaxSatisfied
Step 2.4:             MaxSatisfied = CurrentSatisfied
Step 3:       return MaxSatisfied
*****************************************************************
```

2.2 Heuristic Algorithm

2.2.1 Overview

Heuristic algorithm for Maximum 2-Satisfiability problem is designed to give an efficient solution but this solution is not guaranteed correct solution for a problem. Heuristic algorithms have different types of approaches to find optimal solution such as Simulated Annealing, Tabu Search, Greedy Algorithm and Genetic Algorithm.
To give optimal polynomial time solution in heuristic approach, greedy algorithm is useful. The main aim is to find local optimal solution. And local problem is clause satisfaction in list of clauses. Therefore, each clause needs to be checked. If clause is unsatisfied correct variable value must be assigned to them.

Procedure

- Input: Number of Boolean valued variables and List of clauses
- Output: Number of satisfied clauses
- Initialize variables dictionary.
- Initialize number of satisfied clauses.
- For each clause, check both variables whether they are in dictionary or not.
- If not, assign them to dictionary and their negatives.
- Check satisfaction in clause , if it is satisfied, update Max satisfied number.
- Return number of satisfied clauses.

As a result, the algorithm finds optimal solution locally, but it is not guaranteed to be correct solution.

2.2.2 Pseudocode

Input: n: Number of Boolean valued variables

Clauses: List of clauses with variables for each of clause

Output: Number of Maximum 2-Satisfiability

```
*****************************************************************
Step 1:       Initialize Dictionary assignment_dic
Step 2:       Initialize MaxSatisfied = 0
Step 3:       For current_clause in Clauses
Step 3.1:         x1, x2 = current_clause
Step 3.2:         If  x1 not in assignment_dic
Step 3.2. 1:          Assign x1 to dictionary with correct value
Step 3.2. 2:          Assign negative of x1 to dictionary
Step 3.3.:        If x2 not in assignment_dic
Step 3.3. 1:          Assign x2 to dictionary with correct value
Step 3.3. 2:          Assign negative of x1 to dictionary
Step 3.4.:        If clause is satisfied
Step 3.4. 1:          MaxSatisfied += 1
Step 4:       return MaxSatisfied
*****************************************************************
```

3. Algorithm Analysis

3.1 Brute Force Algorithm

3.1.1 Correctness Analysis

Claim: Given set of m clauses with the k distinct satisfied clauses, MAX 2-SAT algorithm generates all possible combinations for maximize the size of k.

Proof by Induction:

Base Case:

- Boolean variable number is equal to 1 (n = 1).
- Total number of assignments is 2^n ➔ 2^1 ➔ 2.
- Variable ➔ x_1 or $\neg x_1$
- Algorithm needs to check 2 assignment and give number of satisfied clause, which is 1.

  As a result, algorithm correctly determines number of maximum satisfied value in base case.

Inductive Step:

- n = n +1
- Algorithm computes all possible variable combination and total number of assignments is 2^(n+1)
- It is known that algorithm tries to maximize size of k in 2^n assignments so, it will do the same procedure for 2^(n+1) assignments. It will still check every possible combination to maximize k clauses correctly.

As a result, algorithm correctly determines number of maximum satisfied value in inductive step.

3.1.2 Time Complexity

- The algorithm computes every possible variable combination so there are 2^n combinations to compute. ➔ outer loop computes in O(2^n) time.
- Each clause's computation takes constant time, O(1). ➔ Totally m clause so, inner loop computes in O(m) time.
- As a result, total running time is O(2^n . m).
- It can be converted to theta in the same values because it is not a upper bound, it is a running time of algorithm ➔ Θ(2^n. m)

3.1.3 Space Complexity

The algorithm needs to open space for variables, clauses and maximum number of satisfied clauses.

- For n Boolean valued variables ➔ O(n)
- For m number of clauses ➔ O(m)
- Maximum number of satisfied clauses updates itself during execution but deletes the old Max number so, the space for Maximum number satisfied clauses is constant ➔ O(1) space
- In total, space complexity ➔ O(n+m)

3.2 Heuristic Algorithm

3.2.1 Correctness Analysis

Claim: Heuristic MAX 2-SAT algorithm tries to maximize number of satisfied clauses with locally optimal Boolean valued variables.

Number of value combination of variables is equal to 2^n.

Proof by Induction:

Base Case:

- Boolean variable number is equal to 1 (n = 1).
- Number of variables is 2^1 = 2.
- Variable ➔ x_1 or $\neg x_1$
- Algorithm assigns truth value to x_1 for local optimal solution.
- Since there is only one variable, it automatically makes clause satisfied so, heuristic algorithm reached its goal.

Inductive Step:

- Assign n as n+1.
- Total number of variables is 2^(n+1).
- Greedy approach assigns locally optimal value to x_{n+1} variable according to clause’s condition.
- As a result, algorithm still maximizes the number of satisfied clauses. Also, it gives nearly optimal solution in a fast way.

3.2.2 Time Complexity

- First, it creates variable dictionary, in the worst case there will be 2n item in dictionary because their negatives need to be added in dictionary ➔ O(2n) ➔ O(n)
- For loop, it iterates m different clauses, each of them iterated in O(1) constant time so in total ➔ O(m)
- As a result, total time complexity is equal to O(n + m).

3.2.3 Space Complexity

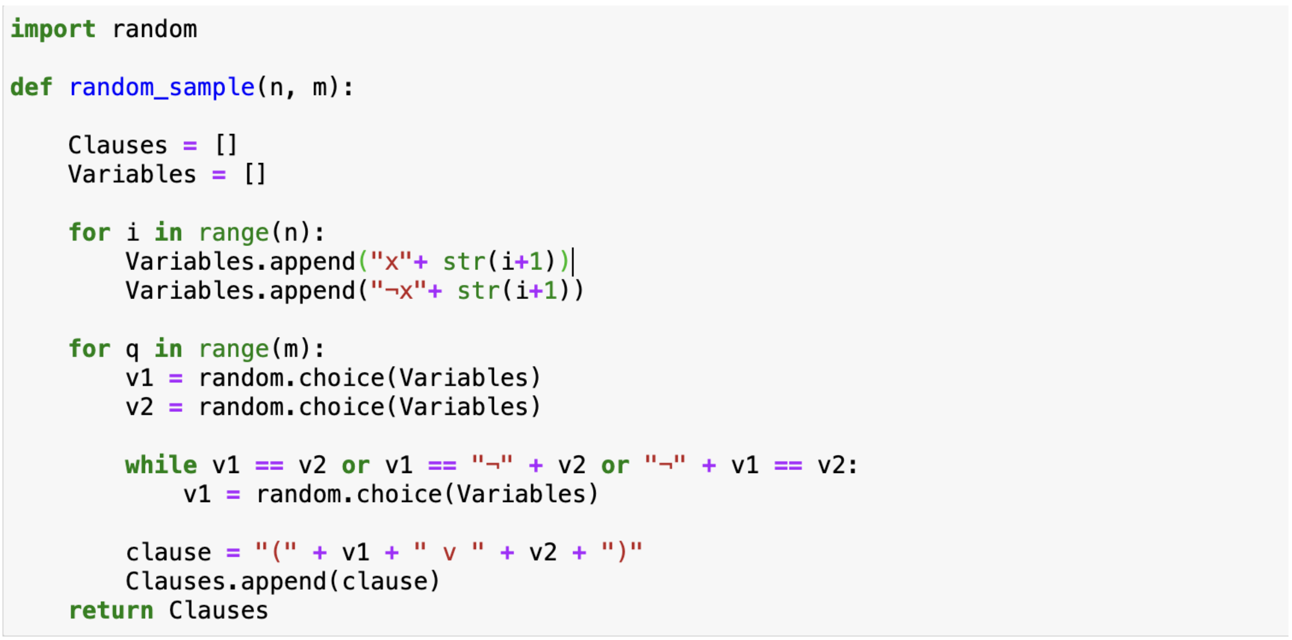
- Algorithm needs space for variable dictionary ➔ O(2n) ➔ O(n)
- Each clause needs to be stored in memory, for each O(1) space needed, In total ➔ O(m)
- As a result, total space complexity is equal to O(n + m).

4. Sample Generation

The random instance generator algorithm is aiming to create random clauses to use in Maximum 2-Satisfiability problem. It takes two inputs, number of Boolean valued variables and number of clauses.

- n = number of Boolean valued variables
- m = number of clauses

- Firstly, algorithm adds Boolean variables and their negations to Variables array.
- In the loop, it chooses two random literals for each clause from Variables
- If literals are same or one is other’s negation it iterates again until find a different variable.
- Then, add that clause to Clause’s array.
- Totally, it iterates m times for m different clause.
- Return Clauses



```python
import random

def random_sample(n, m):

    Clauses = []
    Variables = []

    for i in range(n):
        Variables.append("x"+ str(i+1))
        Variables.append("¬x"+ str(i+1))

    for q in range(m):
        v1 = random.choice(Variables)
        v2 = random.choice(Variables)

        while v1 == v2 or v1 == "¬" + v2 or "¬" + v1 == v2:
            v1 = random.choice(Variables)

        clause = "(" + v1 + " v " + v2 + ")"
        Clauses.append(clause)
    return Clauses
```

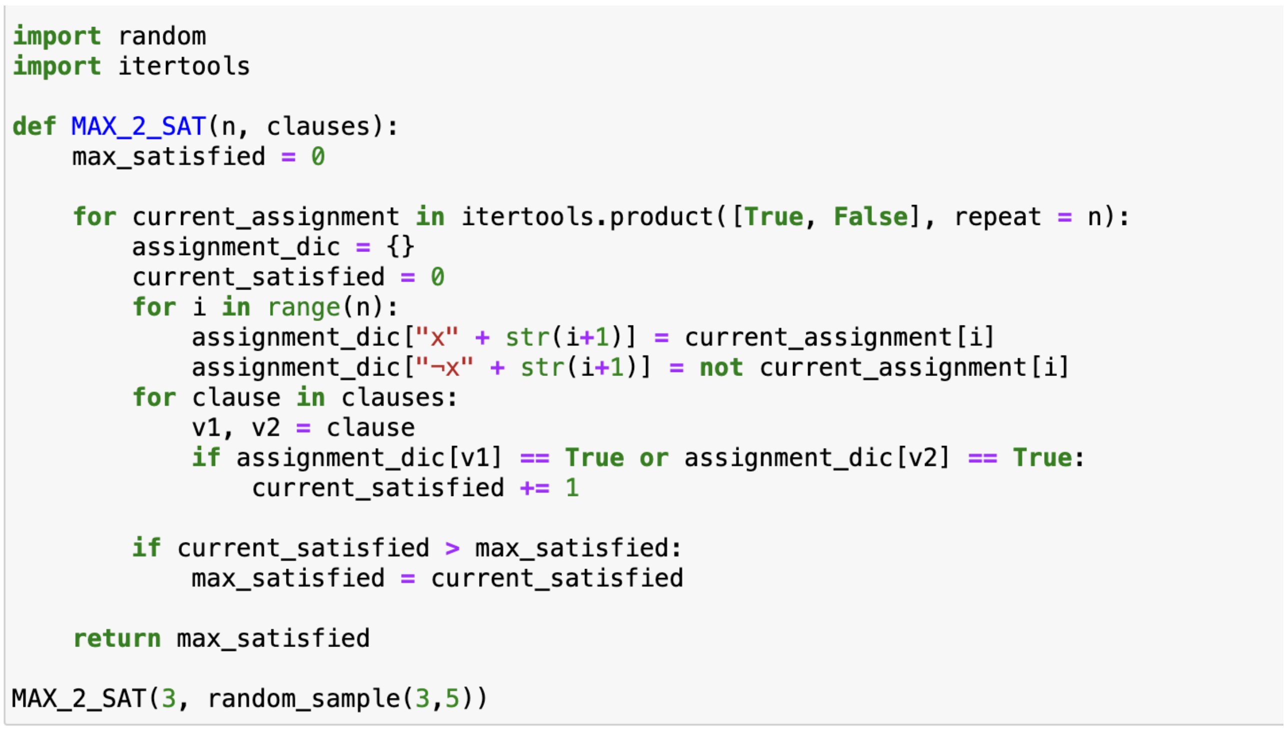
5. Algorithm Implementations

5.1 Brute Force Algorithm

5.1.1 Algorithm

- There are two inputs in algorithm.
- **n:** number of Boolean valued variables.
- **clauses:** It contains all variable combinations of that specific problem.

- Firstly, it assigns zero to max satisfied number.
- In outer loop, it creates every possible “True”, “False” combination with responds to given n number and starts to compute one by one.
- Create dictionary for X_n values, assign values of every variable and also, assign value for every variable’s negation.
- Then, check every clause in clauses for truth values and it is satisfied add 1 to current satisfied number.
- Finally, check if current satisfied number bigger than max satisfied number if it is assigned it as max satisfied number.
- Return max satisfied number.



```
import random
import itertools

def MAX_2_SAT(n, clauses):
    max_satisfied = 0

    for current_assignment in itertools.product([True, False], repeat = n):
        assignment_dic = {}
        current_satisfied = 0
        for i in range(n):
            assignment_dic["x" + str(i+1)] = current_assignment[i]
            assignment_dic["¬x" + str(i+1)] = not current_assignment[i]
        for clause in clauses:
            v1, v2 = clause
            if assignment_dic[v1] == True or assignment_dic[v2] == True:
                current_satisfied += 1

        if current_satisfied > max_satisfied:
            max_satisfied = current_satisfied

    return max_satisfied

MAX_2_SAT(3, random_sample(3,5))
```

5.1.2 Initial Testing of the Algorithm

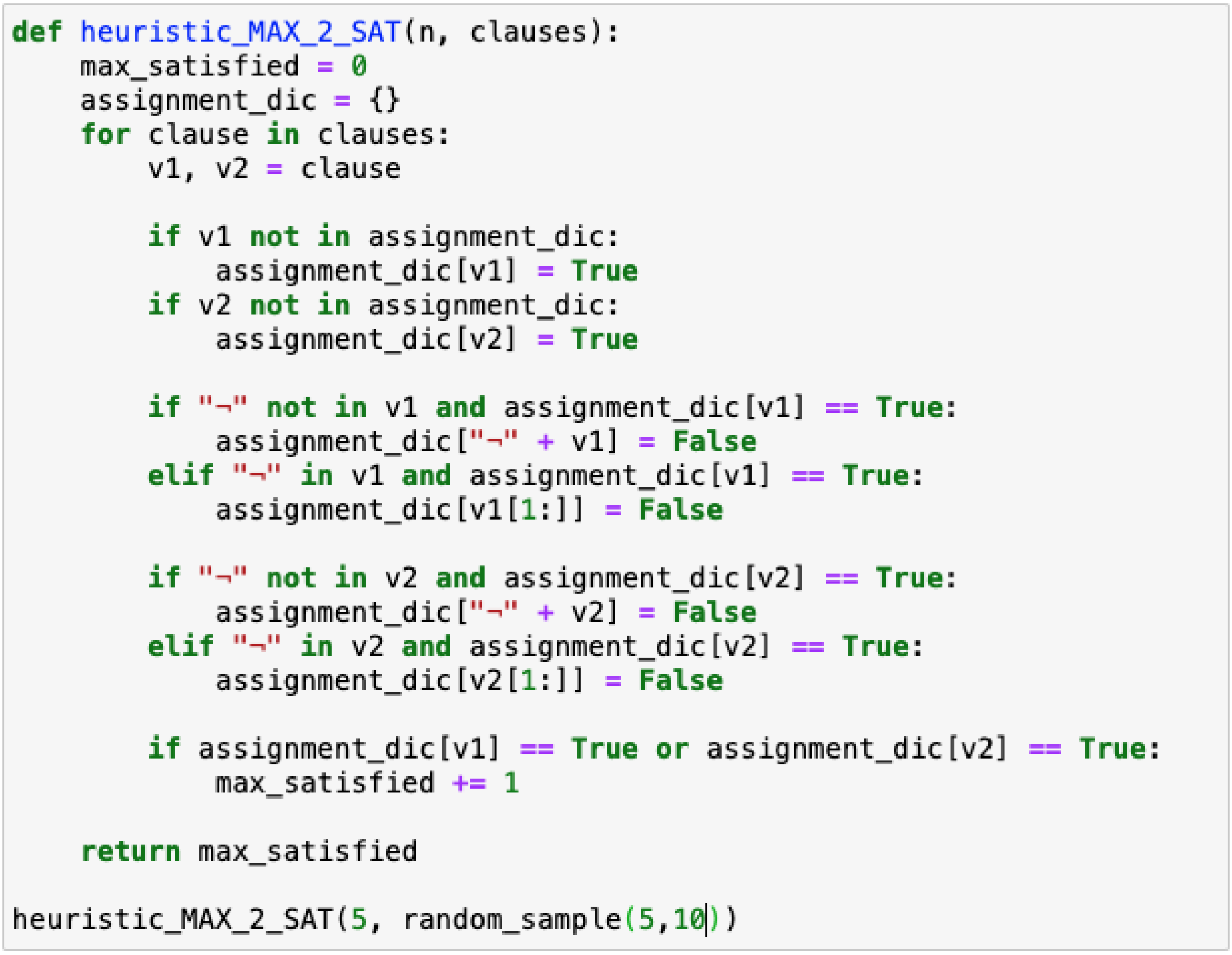
15 number of different variable number(n) and number of clauses(m) is computed. First it computed in Random Sample Generator and then clauses have emerged and used in Brute force algorithm. There were no errors but process getting slower when numbers increased.

Size of the instances (n, m):

(3,5), (3,6), (4,5), (4,6), (4,7), (6,9), (7,11), (5, 10), (8,14), (10, 21), (9,16), (13,25), (15, 29), (12, 32), (16,35)

5.2 Heuristic Algorithm

- In the for loop, algorithm checks the variable whether it is in the dictionary or not.
- If not, it assigns optimal value for its which is ‘True’, and it assigns variable’s negative in dictionary with value ‘False’
- Lastly, it checks clauses satisfaction, if it is satisfied, add one to number of max satisfied clause.



```python
def heuristic_MAX_2_SAT(n, clauses):
    max_satisfied = 0
    assignment_dic = {}
    for clause in clauses:
        v1, v2 = clause

        if v1 not in assignment_dic:
            assignment_dic[v1] = True
        if v2 not in assignment_dic:
            assignment_dic[v2] = True

        if "¬" not in v1 and assignment_dic[v1] == True:
            assignment_dic["¬" + v1] = False
        elif "¬" in v1 and assignment_dic[v1] == True:
            assignment_dic[v1[1:]] = False

        if "¬" not in v2 and assignment_dic[v2] == True:
            assignment_dic["¬" + v2] = False
        elif "¬" in v2 and assignment_dic[v2] == True:
            assignment_dic[v2[1:]] = False

        if assignment_dic[v1] == True or assignment_dic[v2] == True:
            max_satisfied += 1

    return max_satisfied

heuristic_MAX_2_SAT(5, random_sample(5,10))
```

Test Cases

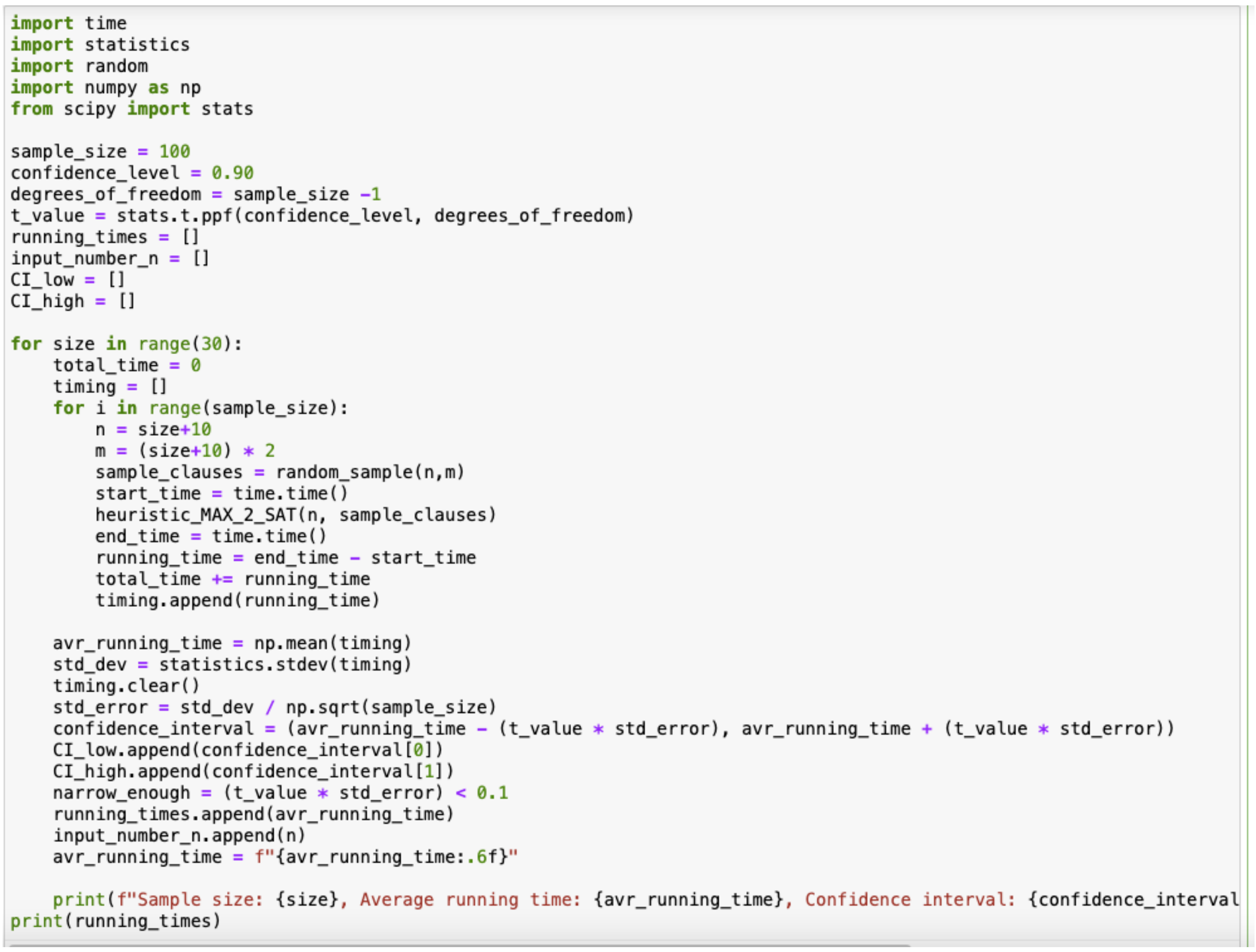
Given table shows Brute-Force algorithm results and Heuristic algorithm results with the given number of Boolean valued variables and number of clauses (n,m).

(n,m)	**Brute-Force algorithm result**	**Heuristic algorithm result**
(3,5)	**5**	**5**
(3,6)	**6**	**5**
(4,5)	**4**	**4**
(4,6)	**6**	**6**
(4,7)	**7**	**7**
(6,9)	**9**	**7**
(7,11)	**11**	**10**
(5,10)	**10**	**7**
(8,14)	**14**	**11**
(10,21)	**21**	**18**
(9,16)	**14**	**13**
(13,25)	**25**	**23**
(15,29)	**28**	**24**
(12,32)	**30**	**24**
(16,35)	**30**	**29**

As a result, Heuristic algorithm gives nearly correct solutions for Maximum 2-Satisfiability problem.

6. Experimental Analysis of The Performance (Performance Testing)

In experimental analysis of the performance part, the aim of the test is to give practical time complexity with the 100 different inputs for 30 different tests. Average running times were calculated for the given tests. Log log plot style used for demonstration of experimental analysis because results did not linearly increased. Also, Confidence intervals are calculated, and results are correctly satisfied the interval condition which is t_value * std_error < 0.1 . Implementation and chart are shown below.



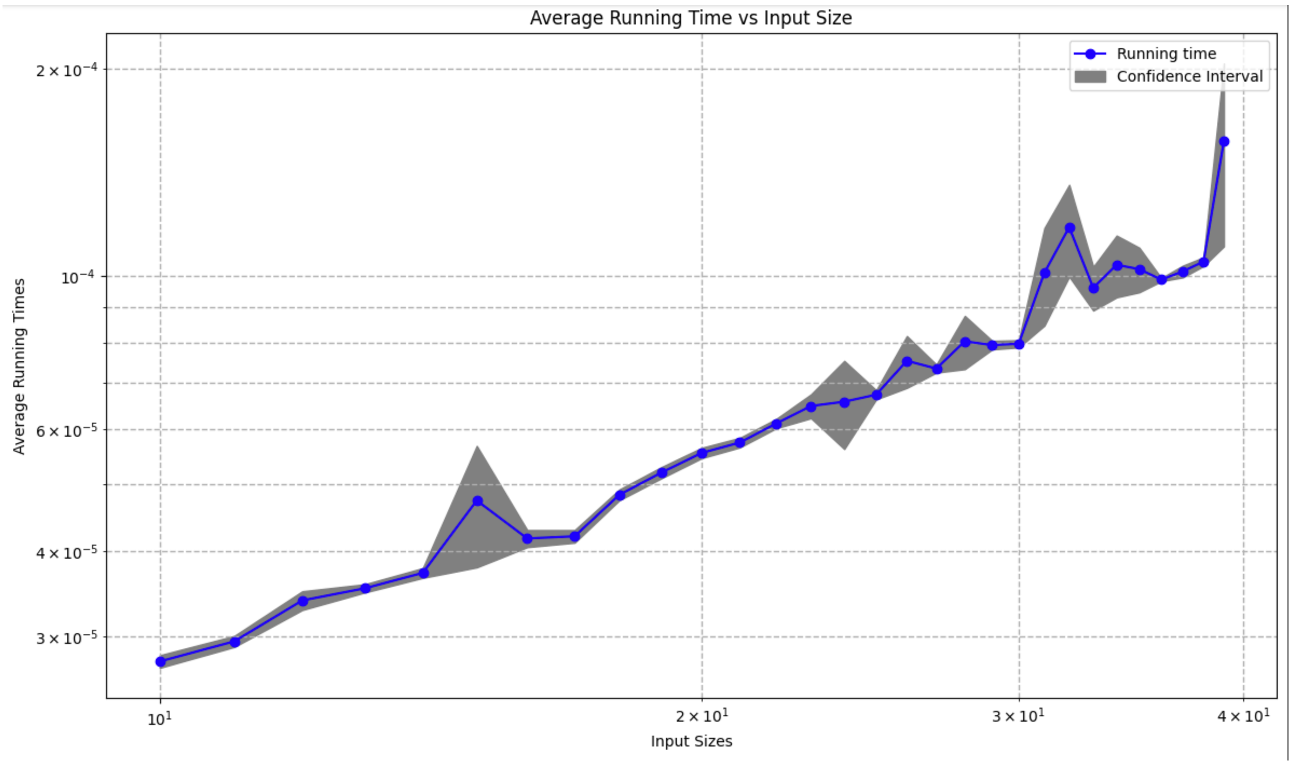
```
import time
import statistics
import random
import numpy as np
from scipy import stats

sample_size = 100
confidence_level = 0.90
degrees_of_freedom = sample_size -1
t_value = stats.t.ppf(confidence_level, degrees_of_freedom)
running_times = []
input_number_n = []
CI_low = []
CI_high = []

for size in range(30):
    total_time = 0
    timing = []
    for i in range(sample_size):
        n = size+10
        m = (size+10) * 2
        sample_clauses = random_sample(n,m)
        start_time = time.time()
        heuristic_MAX_2_SAT(n, sample_clauses)
        end_time = time.time()
        running_time = end_time - start_time
        total_time += running_time
        timing.append(running_time)

    avr_running_time = np.mean(timing)
    std_dev = statistics.stdev(timing)
    timing.clear()
    std_error = std_dev / np.sqrt(sample_size)
    confidence_interval = (avr_running_time - (t_value * std_error), avr_running_time + (t_value * std_error))
    CI_low.append(confidence_interval[0])
    CI_high.append(confidence_interval[1])
    narrow_enough = (t_value * std_error) < 0.1
    running_times.append(avr_running_time)
    input_number_n.append(n)
    avr_running_time = f"{avr_running_time:.6f}"

    print(f"Sample size: {size}, Average running time: {avr_running_time}, Confidence interval: {confidence_interval
print(running_times)
```

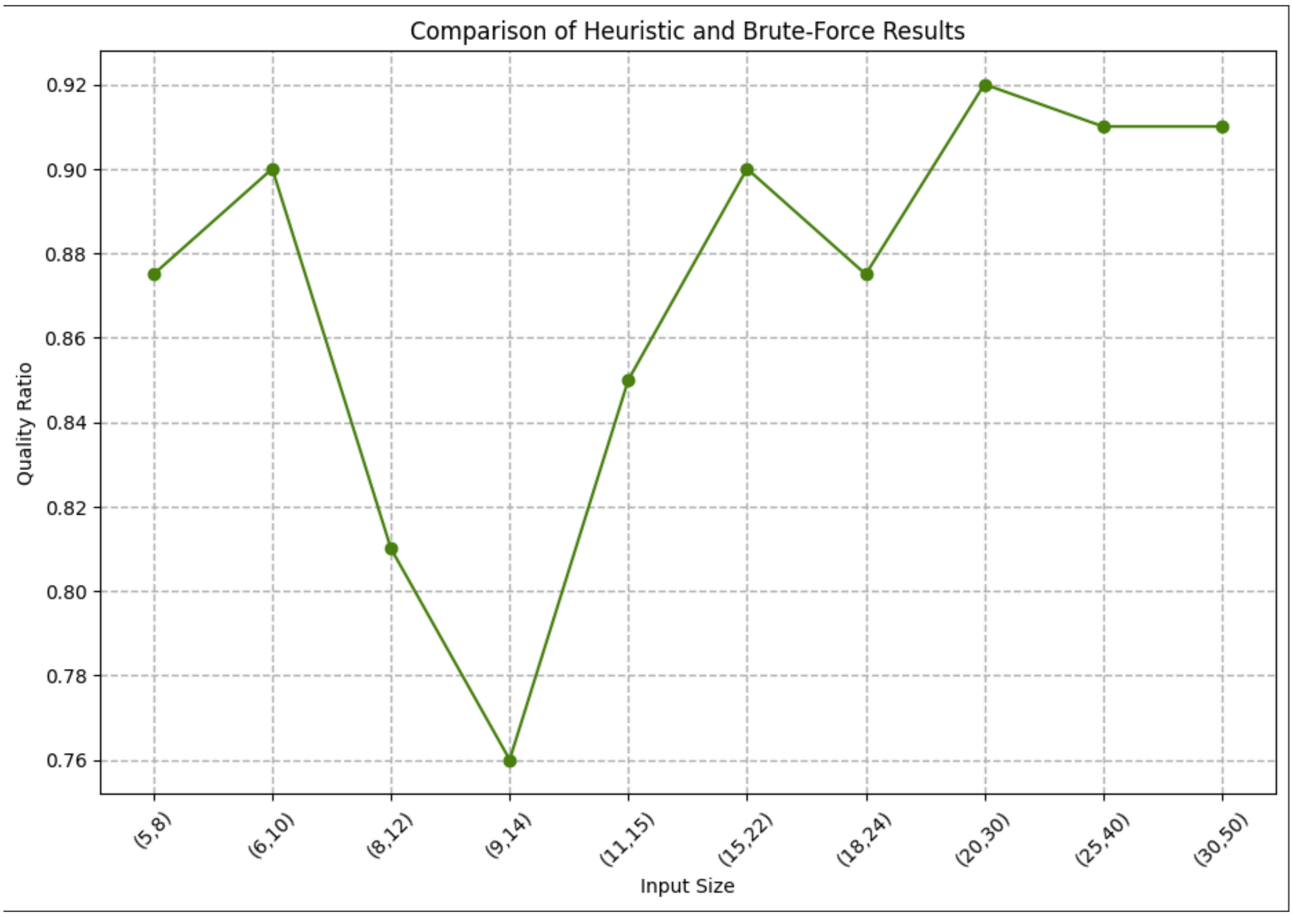


7. Experimental Analysis of the Quality

Input Size (n,m)	Brute-Force Algorithm Result	Heuristic Algorithm Result	Ratio of Heuristic Result / Brute-Force Result
(5,8)	8	7	0.875
(6,10)	10	9	0.9
(8,12)	11	9	0.81
(9,14)	13	10	0.76

(11,15)	14	12	0.85
(15,22)	20	18	0.9
(18,24)	24	21	0.875
(20,30)	28	26	0.92
(25,40)	37	34	0.91
(30,50)	45	41	0.91

Given table shows the Brute-Force algorithm and Heuristic algorithm results to determine Maximum 2-Satisfiability problem solution. According to the quality ratio, heuristic algorithm gives nearly correct result.



When input size increased, we cannot say the quality ratio will monotonically increase or decrease. Because heuristic algorithm determines locally optimal solution, and this solution might be negative for other clauses. Therefore, heuristic result might be very close to exact result or little far from exact result. But if we compute the same input 100 times, we will find the exact result mostly with heuristic algorithm.

8. Experimental Analysis of the Correctness (Functional Testing)

Black Box Testing

Black box testing based on feature of inputs and outputs. It does not consider any data or system's internal workings. In this part, black box implementation will be used to interpret the heuristic Maximum 2-Satisfiability algorithm. Different characteristics test cases are shown below.

Test Case 1: All Clauses Satisfied

Input: n = 3, Clauses = $[\ (x_1 \vee \neg x_3),\ (\neg x_2 \vee x_3),\ (x_1 \vee x_3)\ ]$

Output: {3}

Explanation: First two clauses assign values to all 3 variables so, third clause automatically became satisfied.

Test Case 2: Reverse Clauses

Input: n = 4, Clauses = $[\ (x_1 \lor x_4), (x_2 \lor x_3), (\neg x_1 \lor \neg x_4), (\neg x_2 \lor \neg x_3)]$

Output: $\{2\}$

Explanation: First two clauses assign values to all variables so, other two clauses directly became unsatisfied.

Test Case 3: All literals are positive

Input: n = 3, Clauses = $[\ (x_1 \lor x_2), (x_2 \lor x_3), (x_1 \lor x_3)\ ]$

Output: $\{3\}$

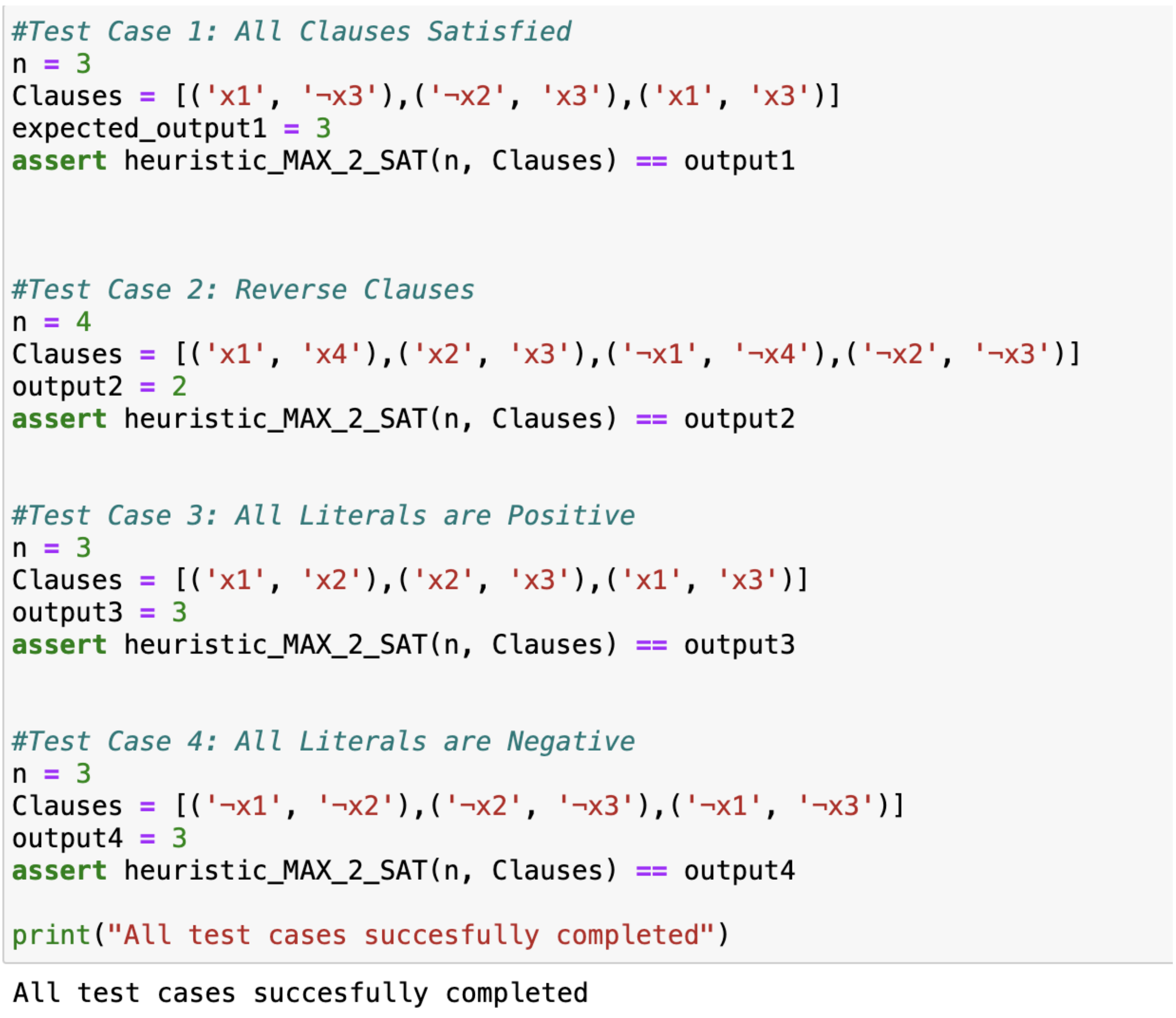
Explanation: All variables are positive, so value combination only contains ‘True’.

Test Case 4: All literals are negative

Input: n = 3, Clauses = $[\ (\neg x_1 \lor \neg x_2), (\neg x_2 \lor \neg x_3), (\neg x_1 \lor \neg x_3)\ ]$

Output: $\{3\}$

Explanation: All variables are negative, so value combination only contains ‘False.



```
#Test Case 1: All Clauses Satisfied
n = 3
Clauses = [('x1', '¬x3'),('¬x2', 'x3'),('x1', 'x3')]
expected_output1 = 3
assert heuristic_MAX_2_SAT(n, Clauses) == output1



#Test Case 2: Reverse Clauses
n = 4
Clauses = [('x1', 'x4'),('x2', 'x3'),('¬x1', '¬x4'),('¬x2', '¬x3')]
output2 = 2
assert heuristic_MAX_2_SAT(n, Clauses) == output2


#Test Case 3: All Literals are Positive
n = 3
Clauses = [('x1', 'x2'),('x2', 'x3'),('x1', 'x3')]
output3 = 3
assert heuristic_MAX_2_SAT(n, Clauses) == output3


#Test Case 4: All Literals are Negative
n = 3
Clauses = [('¬x1', '¬x2'),('¬x2', '¬x3'),('¬x1', '¬x3')]
output4 = 3
assert heuristic_MAX_2_SAT(n, Clauses) == output4

print("All test cases succesfully completed")
```

```
All test cases succesfully completed
```

As a result, black box testing has been verified with the code implementation and all test cases are successfully completed.

White Box Testing

Statement Coverage

Every detail of algorithm needs to checked and computed to successfully complete statement coverage part. In heuristic algorithm, biggest important parts of the code contain if-else conditions. And these statements must be checked for assigning correct Boolean values to variables.

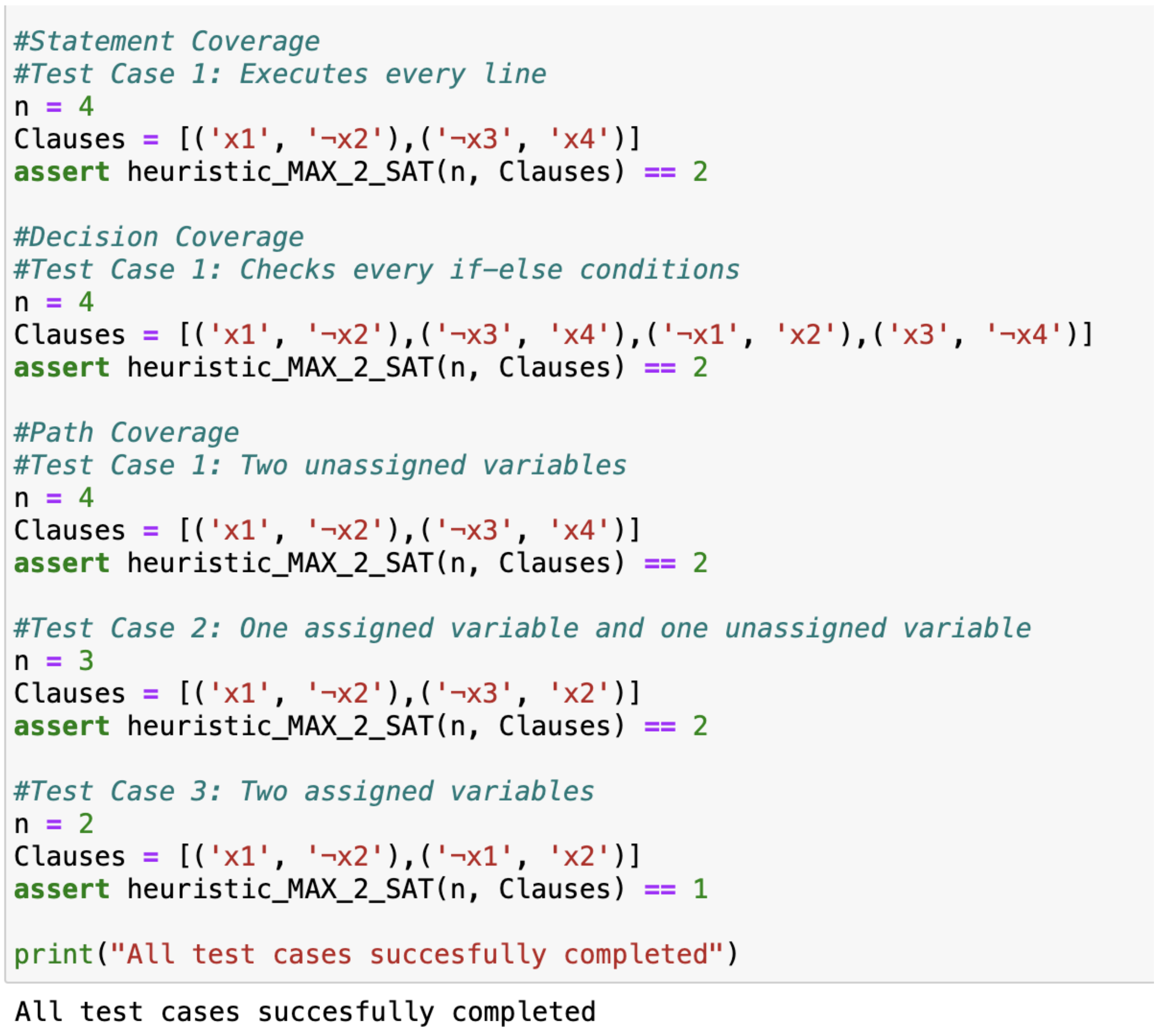
Decision Coverage

We need to be ensured that algorithm makes correct decisions during execution. For that we need to test if-else condition. For example, we need to check if algorithm gives correct decision while variables are unassigned, or one variable is unassigned other one is assigned or both assigned to Boolean values.

Path Coverage

For the correctness of the algorithm, all paths must checked. With different if-else conditions algorithm’s reaction must be tested. Optional algorithm paths are shown below:

- Path with two unassigned variables
- Path with one assigned variable and one unassigned variable
- Path with two assigned variables



```python
#Statement Coverage
#Test Case 1: Executes every line
n = 4
Clauses = [('x1', '¬x2'),('¬x3', 'x4')]
assert heuristic_MAX_2_SAT(n, Clauses) == 2

#Decision Coverage
#Test Case 1: Checks every if-else conditions
n = 4
Clauses = [('x1', '¬x2'),('¬x3', 'x4'),('¬x1', 'x2'),('x3', '¬x4')]
assert heuristic_MAX_2_SAT(n, Clauses) == 2

#Path Coverage
#Test Case 1: Two unassigned variables
n = 4
Clauses = [('x1', '¬x2'),('¬x3', 'x4')]
assert heuristic_MAX_2_SAT(n, Clauses) == 2

#Test Case 2: One assigned variable and one unassigned variable
n = 3
Clauses = [('x1', '¬x2'),('¬x3', 'x2')]
assert heuristic_MAX_2_SAT(n, Clauses) == 2

#Test Case 3: Two assigned variables
n = 2
Clauses = [('x1', '¬x2'),('¬x1', 'x2')]
assert heuristic_MAX_2_SAT(n, Clauses) == 1

print("All test cases succesfully completed")
```

```
All test cases succesfully completed
```

Consequently, white box testing has been verified with the distinct test cases in terms of Statement Coverage, Decision Coverage and Path Coverage.

9. Discussion

In this report, lots of information about Maximum 2-Satisfiability problem obtained. Firstly, brute force algorithm gave us the exact solution without any error. But it causes long process duration because of exponential running time. Therefore, it might not be logical to use it in real life applications like hardware design and Software verification. That’s why heuristic algorithm developed. It does not give the exact solution all the time but running time of this algorithm encourages to use it. Also, when brute force and heuristic results compared the difference between them is around ten percent of the correct solution in large input sizes and it might find the exact solution in small input sizes. For the performance of the algorithm mathematical analyzes were applied. In the performance test of algorithm, there was no input size that the algorithm could not calculate, and it did not give any error. But as the input size increases, the execution time getting bigger. Also, correctness analysis was very detailed and required test cases were implemented considering all types of conditions. Consequently, experimental analysis and theoretical analysis are compatible with each other. And with the enhanced random sample generator the algorithm can be more helpful in the future.

References

Garey M., Johnson D., and Stockmeyer L. Some simplified NP-complete graph problems. Theor. Comput. Sci. 1:237–267, 1976.