Assignment 1: The classical combinatorial optimization problem- MAXSAT

Ali Ahammad

a81317@ualg.pt

Course: Metaheuristics

Answers in order:

Task 1:

The DIMACS CNF format is a textual representation of a formula in conjunctive normal form. A formula in conjunctive (logical and) of a set of clauses. Each clause is a disjunction (logical or) of a set of literals. A literal is a variable or a negation of a variable. DIMACS CNF uses positive integers to represent variables and their negation to represent the corresponding negated variable.

If we let x_i represent variables that an assume only the values *true* or *false*, then a sample formula in conjunctive normal form would be

$$(x_1 \vee x_3 \vee \neg x_4) \wedge (x_4) \wedge (x_2 \vee \neg x_3)$$

where $\vee$ represents the *or* Boolean connective, $\wedge$ represents *and* and $\neg x_i$ is the negation of x_i.

Given a set of clauses $C_1, C_2, \ldots., C_m$ on the variables $x_1, x_2, \ldots., x_n$, the satisfiability problem is to determine if the formula

$$C_1 \wedge C_2 \wedge \ldots \wedge C_m$$

is satisfiable. That is, is there an assignment of values to the variables so that the above formula evaluates to *true*. Clearly, this requires that each C_j evaluate to *true*.

The maximum satisfiability problem is to find an assignment of values to the variables to have the maximum number of C_j evaluate to true. To represent an instance of such problems, we will create an input file that contains all the information needed to define a satisfiability problem or a maximum satisfiability problem.

Task 2:

My hoos.cnf is as followed according to task 2 formula:

```
c This Formular is generated by mcnf
c
c     horn? no
c     forced? no
c     mixed sat? no
c     clause length = 3
c
p cnf 5  6
-1 2 0
-2 1 0
-1 -2 -3 0
 1 2 0
-4 3 0
-5 3 0
%
0
```

Task 3:

This assignment is to be solved with exhaustive brute force. For each combination of *True/False* values (called an "assignment"), the program checks how many clauses are satisfied. A clause is satisfied if at least one of its variables is True when it is supposed to be.

- If a variable appears without a minus sign (e.g., "1"), it means it must be True for the clause to be satisfied.
- If a variable appears with a minus sign (e.g., "-2"), it means that variable must be False for the clause to be satisfied.

The program goes through each clause, checks the assignment for that clause, and counts how many clauses are satisfied. I am using pseudocode to explain the algorithm I have used to solve the task:

```
1. Read the input DIMACS CNF file:
    - Open the file
    - Initialize clauses, number of variables, and number of clauses
    - For each line in the file:
        if the line is a comment, empty, or contains '%' or '0', skip it
        if the line starts with 'p', parse the number of variables and clauses
        if the line contains a clause, convert it to a list of integers (literals) and add it to
the clauses list
    - Ensure the number of parsed clauses matches the expected number of clauses
```

```
2. Initialize variables for finding satisfying assignments:
```

```
    - best_assignments = an empty list to store valid assignments
    - satisfied_assignment_count = 0 to count the number of satisfying assignments

3. Generate all possible truth assignments for the variables:
    for each possible combination of true/false assignments for all variables:
        Initialize satisfied_clauses = 0 to count how many clauses are satisfied by this
assignment

        For each clause:
            Initialize satisfied = false
            For each literal in the clause:
                - Extract the variable (absolute value of literal)
                - Check if the literal is satisfied based on its sign and the current truth
assignment
                if the literal is satisfied:
                    set satisfied = true and break out of the loop

            if the clause is satisfied, increment satisfied_clauses
            else, break out of the loop (the current assignment is invalid for MAXSAT)

        if the current assignment satisfies all clauses:
            Increment satisfied_assignment_count
            Add the current assignment to best_assignments

4. Output the results:
    - If there are satisfying assignments:
        print the total number of satisfying assignments
        for each satisfying assignment, print its values
    - If no satisfying assignment is found:
        print a message indicating no solution

5. Print the execution time
```

Task 4:

When I run my program for hoos.cnf, I have the following result:

```
Number of variables: 5
Number of clauses: 6
Number of satisfying assignments: 1
Best assignments:
No 1 best assignment: True, True, False, False, False
Execution time: 0.000000 seconds
```

Task 5:

When I run my program for uf20-01.cnf, I have the following result:

```
Number of variables: 20
Number of clauses: 91
Number of satisfying assignments: 8
Best assignments:
No 1 best assignment: False, True, True, True, False, False, False, True, True, True, True,
False, False, True, True, False, True, True, True, True
No 2 best assignment: True, False, False, False, False, True, False, False, False, False, False,
False, True, True, True, False, True, False, False, True
No 3 best assignment: True, False, False, False, False, True, False, False, True, False, False,
False, False, True, True, False, True, False, False, True
No 4 best assignment: True, False, False, False, False, True, False, False, True, False, False,
False, True, True, True, False, True, False, False, True
No 5 best assignment: True, False, False, True, False, False, False, False, False, True, False,
False, True, True, True, False, True, False, False, True
No 6 best assignment: True, False, False, True, False, False, False, True, False, True, False,
False, True, True, True, False, True, False, False, True
No 7 best assignment: True, False, False, True, False, True, False, False, False, False, False,
False, True, True, True, False, True, False, False, True
No 8 best assignment: True, False, False, True, False, True, False, False, False, True, False,
False, True, True, True, False, True, False, False, True
Execution time: 1.219740 seconds
```

Task 6:

To estimate the time needed for the SAT solver to complete the task on an instance like my code, I need to break down the complexity of the algorithm and consider the factors involved:

- **Number of Variables (n):** the brute force algorithm generates all possible truth assignments for n variable. For each variable, there are two values (*True or False*), so the total number of truth assignments is (2^n).
- **Number of clauses (m):** for each truth assignment, the algorithm must evaluate whether it satisfies all *m* clauses. This involves checking each clause and its literals.
- The time complexity is proportional to both the number of truth assignments and the number of clauses that must be checked per assignment.

So, the total time complexity could be determined by

- Evaluating all truth assignments: (O(2^n))
- Checking each clause: (O(m))

- Total complexity: (O(2^n * m))

Estimation based on Problem size:

To estimate the time required, I need to know the values of n (number of variables) and m (number of clauses) in your instance. Let's assume:

- (n=100) (suppose in the uf100-0430.cnf file)
- (m=430) (suppose in the same file)
- Number of truth assignments: ($2^{100} \sim 1.27 * 10^{30}$)
- For each assignment, I need to check all 430 clauses.
- Therefore, in the worst case, we have to perform ($2^{100} \sim 1.27 * 10^{30}$*430~$5.45 * 10^{32}$)

To the estimate the runtime, I needed to consider the processor speed which is assumed to be in typical performance around 3 GHZ, meaning 3*10^9 cycles per second.

Assuming that each clause check requires a few CPU cycles (let's estimate per clause check on average), we get:

1) Operations per second (OPS):

   3*$10^9 \frac{cycles}{second} * 10 \frac{operations}{cycle} = 3 * \frac{10^{10} operations}{second}$

2) Total time required:

   Time=$\frac{5.45*10^{32} operations}{3*10^{10}\ operations/second}$~1,82*$\ 10^{22} seconds$

3) Converting seconds to years:

$$\frac{1.82 * 10^{32}\text{seconds}}{60 * 60 * 24 * 365} \sim 5.77 * 10^{14}\text{years}$$

So solving uf100-0430.cnf in my processor is computationally impossible for large values of n like 100.