Universitat de Lleida

University of Lleida

Practice 2

MaxSAT

Artificial Intelligence

Pere Muñoz Figuerol

December 2022

Contents

1 Introduction

In this practice we will work with a MaxSAT solver, with the objective of understanding how it works and how to use it.
First we will model some graph problems in MaxSAT, like the *Minimum Vertex Cover*, the *Maximum Clique* and the *Max-Cut* problems.
Secondly, we will implement the transformation of a *MaxSAT* formula to a (1,3) *Weighted Partial MaxSAT* formula.
Finally, we will model the *Combinatorial Actions* problem in MaxSAT, and we will solve it translating it to a (1,3) *Weighted Partial MaxSAT* formula.

2 Graph Problems

In this section we will see the modeling of the *MaxSAT* graph problems seen at the Introduction.
First we will model the *Minimum Vertex Cover* problem.
Then we will model the *Maximum Clique* problem.
And finally we will model the *Max-Cut* problem.

2.1 Minimum Vertex Cover

The *Minimum Vertex Cover* problem is a graph problem that consists in finding the minimum subset of vertices of a graph such that each edge of the graph is incident to at least one vertex of the subset.
It is NP-complete, so we will model it in *MaxSAT*, concretely in *Weighted Partial MaxSAT* because we have soft and hard clauses with weights.
First we initialize a new formula, and we add one variable to it for each vertex of the graph.
Then we create a soft clause for every node of the graph, with weight 1. The variable representing each vertex will be negated in the clause.

```
for n in nodes:
    formula.add_clause([-n], weight=1)
```

After that, we create a hard clause for every edge of the graph.

```
for n1, n2 in edges:
    formula.add_clause([nodes[n1 - 1], nodes[n2 - 1]])
```

Finally, we solve the formula and we print the solution.

2.2 Maximum Clique

The *Maximum Clique* problem is a graph problem that consists in finding the maximum subset of vertices of a graph such that each pair of vertices of the subset is connected by

an edge.
It is NP-complete, so we will model it in *MaxSAT*, concretely in *Weighted Partial MaxSAT* because we have soft and hard clauses with weights.
Same as the previous problem, we initialize a new formula, and we add one variable to it for each vertex of the graph.
Then we create a soft clause for every node of the graph, with weight 1. This time the variable representing each vertex will not be negated in the clause.

```
for n in nodes:
    formula.add_clause([n], weight=1)
```

After that, we create a hard clause for every edge that doesn't exist in the graph. The variables representing the vertices of the inexistent edge will be negated in the clause.

```
for n1 in nodes:
    for n2 in nodes:
        if (n1, n2) not in self.edges and
        (n2, n1) not in self.edges and n1 < n2:
            formula.add_clause([-nodes[n1-1], -nodes[n2-1]])
```

As an appreciation, we can see that we are only adding the clause if the first node is smaller than the second one. This is for avoiding adding the same clause twice, because the graph is undirected.
Finally, we solve the formula and we print the solution.

2.3 Max-Cut

The *Max-Cut* problem is a graph problem that consists in finding a partition of the vertices of a graph into two disjoint subsets such that the number of edges between the two subsets is maximized.
It is NP-complete, so we will model it in *MaxSAT*, concretely in *Weighted Partial MaxSAT* because we have soft and hard clauses with weights.
Same as the previous problem, we initialize a new formula, and we add one variable to it for each vertex of the graph.
Then we create two soft clauses for every edge of the graph, with weight 1.

```
for n1, n2 in self.edges:
    formula.add_clause([nodes[n1-1], nodes[n2-1]], weight=1)
    formula.add_clause([-nodes[n1-1], -nodes[n2-1]], weight=1)
```

As in this problem we only have soft clauses, at this point we can solve the formula and print the solution.

3 Transformation to (1,3) Weighted Partial MaxSAT

In this section we will see the transformation of a *MaxSAT* formula to a (1,3) *Weighted Partial MaxSAT* formula.
The method that we are going to implement is the `to_13wpm()` function, that receives a *WCNF* formula and returns the equivalent (1,3) *Weighted Partial MaxSAT* formula.
First we initialize a new formula, and we add one variable to it for each variable of the original formula, because these are the minimum variables that our new formula will have.

3.1 Transformation of soft clauses

For the soft clauses transformation, we will create an additional variable for each soft clause of the original formula. This variable will help us to represent the literals of the original clause. Then, we will add a soft clause, composed with the additional variable negated and with the same weight of the original formula.
After that, we will call the `to_13wpm_hard()` function, that converts the hard clause to a *CNF-3* formula (explained in the next section), with a clause composed with the addicional variable and the literals of the original soft clause.

```
for clause, weight in self.soft_clauses:
    new_hard = [[new_var] + clause]
    formula13.to_13wpm_hard(formula13, new_hard)
```

3.2 Transformation of hard clauses

For the hard clauses transformation, we will check if the clause is a 3-CNF clause. If it is, so it has 3 literals, we will add it to the new formula directly. If it is not, so it has less or more than 3 literals, we will transform it to a 3-CNF clause, and then we will add it to the new formula.

For this transformation, I have implemented an auxiliary function called `to_13wpm_hard()`, that receives a *WCNF* formula and a list of clauses, and it adds the *CNF-3* hard clauses equivalent to the formula.

The first step of the transformation is to check if the clause is a 3-CNF clause. If it is, so it has 3 literals, we will add it to the new formula directly.

```
if len(clause) == 3:
    formula13.add_clause(clause, TOP_WEIGHT)
```

If it is less than 3 literals, we have to duplicate the literals of the clause until it has 3 literals.

```
elif len(clause) < 3:
    extended_clause = clause + clause[:3 - len(clause)]
    formula13.add_clause(extended_clause, TOP_WEIGHT)
```

If it is more than 3 literals, we have to transform it to a 3-CNF clause concatenating the literals of the clause in groups of 3, using some auxiliary variables.

```
else:
    new_var = formula13.new_var()
    sub_clause = clause[:2] + [new_var]
    formula13.add_clause(sub_clause, TOP_WEIGHT)
```

Then, we will call the function recursively with the auxiliary variable negated and the rest of the literals of the clause.

```
formula13.to_13wpm_hard(formula13, [[-new_var] + clause[2:]])
```

4 *Combinatorial Auctions* modelling as *Weighted Partial MaxSAT*

In this section we will see the modelling of the *Combinatorial Auctions* problem as a *Weighted Partial MaxSAT* formula.
First of all, I have implemented a class called `CombinatorialAuction`, that represents a *Combinatorial Auction* problem.
This class has as attributes all the agents and goods declared in the input file and a dictionary of bids of type `Bid`.
This class `Bid` has as attributes the agent that makes the bid, the goods that the agent wants to buy and the price that the agent is willing to pay.
With this structure defined, we can start modelling the problem.
The most important method of the `Combinatorial Auction` class is the `solve()`, which solves the problem and prints the solution.
This method will call the `toWPMS()` function, that transforms the *Combinatorial Auction* problem to a *Weighted Partial MaxSAT* formula.
Then, it will call the `checkSolution()` function, that checks if the solution is valid.
All the modelling is done in the `toWPMS()` function, and I have followed the steps that we saw in class.
As I have to redact a short report, I will not explain all the steps of the modelling, but I will explain why I decided to store the bids in a dictionary and the `is_compatible()` function of the `Bid` class.

4.1 Storing the bids

The first decision of the modelling that I will explain is the reason of the storage of the bids in a dictionary. The key of the dictionary will be the *MaxSAT* variable assigned to the clause representing that specific bid, and the value will be an object of type `Bid`. This way, we can easily interpret the solution of the *Weighted Partial MaxSAT* formula, and translating it to our specific problem.

4.2 Checking if a bid is compatible with another

The second decision that I will explain is how I check if a bid is compatible with another. This is done in the `is_compatible()` function of the `Bid` class. This function receives a bid, specified the *Other* bid, and it checks if the *Other* bid is compatible with the current bid. Concretely, we have to check that the goods of the *Other* bid and the current bid are different.
For doing so, first we store the goods of the bids in two different sets. Then, we check if the intersection of the sets is empty using the `isdisjoint()` function of the `set` class, and we return the result.

```
def is_compatible(self, other):
    self_goods = set(self.goods)
    other_goods = set(other.goods)
    return self_goods.isdisjoint(other_goods)
```

I think that doing it this way is more efficient than checking it with other methods like `intersection()`, because the `isdisjoint()` function stops its runtime when finding a match in the two sets, and the `intersection()` function creates a new set with the intersection so it has to compute all the two sets.

5 Conclusions

This practice has been very interesting, because I have learned a lot about *Weighted Partial MaxSAT*, a topic that was unknown for me, how to model some graph and real-life problems as this notation in Python.