Second Assignment Artificial Intelligence 2022-2023

Departament d'Informàtica i Enginyeria Industrial
Universitat de Lleida
carlos@diei.udl.cat
eduard.torres@udl.cat
josep.alos@udl.cat

1. Statment

The goal of this assignment is to evaluate the knowledge of the student regarding the modelling of optimization problems as Maximum Satisfiability (MaxSAT) formulas. You will find all the required files that you will need to implement this assignment in the virtual campus, under the `Labs/maxsat` folder.

1.1. Graph problems modelling: 2 points

We have shown in the laboratory classes how to translate the Minimum Vertex Cover problem to a MaxSAT formula. Following the same approach you must now translate the Maximum Clique and Max-Cut problems. You can find their descriptions in the slides presented in class.

The functions that you have to implement (`min_vertex_cover`, `max_clique` and `max_cut`) can be found in the *graph.py* file.

1.2. (1,3) Weighted Partial MaxSAT transformation: 1 point

Implement the procedure `to_13wpm` in the file *wcnf.py*. This procedure must transform any MaxSAT formula to a (1,3) Weighted Partial MaxSAT formula.

1.3. *Combinatorial Auctions* modelling as WPMS: 7 points

Implement an application that solves the **Combinatorial Auctions** problem by translating it to Weighted Partial MaxSAT.

1.3.1. Tasks to be performed

You will have to provide a Python file called *auct_ solver.py*. This Python file must be able to solve the Standard formulation of the Combinatorial Auctions problem, as well as the Extended formulation (see section 1.3.4).

In particular, your application must fulfil the following requirements:

- Parse the input arguments of your application (see section 1.3.2).
- Parse the input file (specified in section 1.3.3).
- Translate the problem to a Weighted Partial MaxSAT formula following the transformations that you can find in the slides.
- Solve the formula using a MaxSAT solver.
- Translate the solution returned by the MaxSAT solver into the **Combinatorial Auctions** problem, and return as result the benefit for the auctioneer and the list of goods obtained by each agent (for more information see section 1.3.5).
- Ensure the returned solution is a valid solution to the input Combinatorial Auction problem (see section 1.3.6).

1.3.2. Application arguments

Your *auct_solver.py* script must receive the following arguments:

1. The path to a MaxSAT solver executable that you will use in your application to solve the given problem.
2. The path to the input instance of the Combinatorial Auctions problem.
3. An optional flag `--no-min-win-bids` that will deactivate the *Minimum winning bids* constraints (see section 1.3.4).

This would be an example execution considering the *Minimum winning bids* constraints:

```
python3 auct_solver.py ./MyMaxSAT inst.auct
```

This would be an example execution **deactivating** the *Minimum winning bids* constraints:

```
python3 auct_solver.py ./MyMaxSAT inst.auct --no-min-win-bids
```

1.3.3. Input instance format

You can find some instances of the **Combinatorial Auctions** problem in the folder `Labs/maxsat` of the virtual campus, which you must use to check your implementation. These instances have the following format:

1. `a <agent-name-1> <agent-name-2> ... <agent-name-n>`: Definition of the different agents that participate in the Combinatorial Auction. **This must be the first line of the file. You can assume all the agents' names are unique.**
2. `g <good-name-1> <good-name-2> ... <good-name-n>`: Definition of the different goods of the Combinatorial Auction. **This must be the second line of the file. You can assume all the goods' names are unique.**
3. `<agent-name> <good-name> [good-name ...] <price>`: Specification of a bid. The agent `<agent-name>` bids for the set of goods `<good-name> [good-name ...]` with a price of `<price>`. **You can assume the agent and goods have been already declared, and the set of goods in the bid are unique.**

   There can be as many bids as needed, each of one defined in a different line of the file. Notice that each agent can also have more than one bid.

Consider the following example:

```
a a1 a2 a3
g g1 g2 g3 g4
a1 g1 g2 20
a2 g2 5
a2 g3 10
a2 g4 20
a3 g1 15
a3 g4 10
```

In this example we have 3 agents $\{a1, a2, a3\}$, 4 goods $\{g1, g2, g3, g4\}$ and 6 bids. In the first bid, agent $a1$ is bidding for the goods $\{g1, g2\}$ with a price of 20.

1.3.4. Encoding the *Combinatorial Auction* problem into Weighted Partial MaxSAT

To encode the *Combinatorial Auction* problem into Weighted Partial MaxSAT you must implement the constraints of the Standard Formulation that are presented in the slides (see slides 28-29). You can consider that two bids are incompatible if they have some good in common.

Regarding the *Minimum winning bids* constraints that are part of the Extended formulation of the problem (see slide 30), you must provide an option to the user to deactivate them through the usage of the `--no-min-win-bids` flag (see section 1.3.2). These constraints enforce that at least one of the bids of an agent must be a winning bid, which might be too restrictive for certain scenarios.

By allowing the inclusion and exclusion of the *Minimum winning bids* constraints we are providing solutions for two variations of the same problem.

1.3.5. Output solution format

Your application must print in stdout the following string as solution:

```
Benefit: <benefit>
<winning-bid-1>
<winning-bid-2>
...
```

The benefit corresponds to the sum of the prices of the winning bids. Each winning bid must be printed following this format:

`<agent-name>: <good-name-1>,<good-name-2>,...,<good-name-n> (Price <price>)`.

As an example, this would be one possible solution for the instance presented in section 1.3.3:

```
$ python3 auct_solver.py ./MyMaxSAT ./inst.auct
Benefit: 40
a1: g1,g2 (Price 20)
a2: g3 (Price 10)
a3: g4 (Price 10)

$ python3 auct_solver.py ./MyMaxSAT ./inst.auct --no-min-win-bids
Benefit: 50
a2: g2 (Price 5)
a2: g3 (Price 10)
a2: g4 (Price 20)
a3: g1 (Price 15)
```

1.3.6. Validation of the solution

Given a solution to the *Combinatorial Auctions* problem you must ensure that this solution corresponds to a valid one. To achieve this you must check:

1. That there are no conflicts between the goods of the selected bids.
2. That for each agent, at least one of their bids is a winning bid **(only when *Minimum winning bids* constraints are active)**.

As output, right after printing the solution, you will need to specify whether this solution if valid of not by following this format:

```
[Valid|Invalid] Solution
```

1.4. Implementation

We will consider the quality and efficiency of the implemented algorithms. It is required to follow these steps for your implementation:

- The programming language is Python 3.
- The usage of Object-Oriented Programming.
- Simplicity and readability of your code.
- It is recommended to follow the style guide [1].

1.5. Documentation

Pdf report (max ≈ 4 pages) with:

- a description of the design decisions in your implementation of the algorithms

[1] https://www.python.org/dev/peps/pep-0008/

The first page of the report must include the name of all the members of your group. The assignment can be done in groups of two students or individually.

We will consider the writing and presentation of the report.

1.6. Delivery

The delivery must contain:

- All the **modified or added** source code files.
- Assignment report.

All the required content must be delivered in a compressed file with the name `ia-prac2.[tgz|tar.gz|zip]`

```
ia-prac2.[tgz|tar.gz|zip]
├── src
│   ├── graph.py
│   ├── auct_solver.py
│   └── wcnf.py
└── report.pdf
```

NOTE: Take into account that you will need the file **msat_runner.py** and a MaxSAT solver (in our case EvalMaxSAT) to develop this assignment, but you do NOT have to include these files in the deliverable.