Lab 2: Maximum Satisfiability (MAXSAT)[1]

Released: Monday 19th February, 2018
Deadline: February 23, 2018
Weight: 10 % for 06-27819, or 11.25 % for 06-27818 %

You need to implement one program that solves Exercise 1-3 using any programming language. In Exercise 5, you will run a set of experiments and describe the result using plots and a short discussion.

(In the following, replace `abc123` with your username.) You need to submit one zip file with the name `niso2-abc123.zip`. The zip file should contain one directory named `niso2-abc123` containing the following files:

- the source code for your program
- a Dockerfile (see instructions in the appendix)
- a PDF file for Exercises 4 and 5.

Maximum satisfiability (MAXSAT) is a fundamental combinatorial optimisation problem where we need to assign truth values to a set of variables. Before we can define the objective value of an assignment, we need to introduce some terminology.

An *assignment* of truth values to n variables can be represented as a bitstring $x \in \{0,1\}^n$ of length n, where a 1-bit in position i, i.e. $x_i = 1$, indicates that we assign the value TRUE to the variable x_i. Conversely, a 0-bit in position i, i.e., $x_i = 0$, indicates that we assign the value FALSE to the variable x_i. A *positive literal* is a variable x_i, for any[2] $i \in [n]$, and a *negative literal* is the negation $\neg x_i$, for any $i \in [n]$. A *clause* is a set of one or more literals. A clause is *satisfied* by an assignment $x \in \{0,1\}^n$ if it contains at least one positive literal x_i such that $x_i = 1$, or at least a negative literal $\neg x_i$ such that $x_i = 0$. An *instance* of the MAXSAT problem consists of a set of m clauses, over n variables. The objective of the MAXSAT problem is to find an assignment which maximises the number of satisfied clauses. This is an NP-hard problem[3].

The following example is an instance of the MAXSAT problem with $n = 3$ variables and $m = 2$ clauses.

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

The assignment $x = 000$ satisfies the first clause $(\neg x_1 \vee x_2 \vee x_3)$ because of the negative literal $\neg x_1$, but not the second clause $(x_1 \vee x_2 \vee x_3)$ which contains only positive literals. In contrast, the assignment $x = 011$ satisfies both clauses.

[1] Revised: Monday 19th February, 2018 at 23:34.
[2] Here, we use the notation $[n] := \{1, \ldots, n\}$.
[3] If you can design an efficient algorithm for this problem, you can claim a US $ 1 million prize from the Clay Mathematics Institute. `http://www.claymath.org/millennium-problems`

Exercise 1. *(10 % of the marks) Implement a satisfiability checker which given a clause and an assignment determines whether the clause is satisfied by the assignment.*

A clause is represented by a sequence of values, where you can ignore the first value. Each of the next values is either a positive integer, to indicate a positive literal, or a negative integer, indicating a negative literal. The indices of the literal start from 1, and not 0. The representation of a clause is completed by a integer `0`*. As an example, the string* `0.5 2 1 -3 -4 0` *represents the clause* $(x_2 \vee x_1 \vee \neg x_3 \vee \neg x_4)$.

Input arguments:

- `-assignment` *an assignment as a bitstring*
- `-clause` *a clause description, as specified above*

Output:

- `1` *if the clause is satisfied, and* `0` *otherwise*

Example:

```
[pkl@phi ocamlec]$ app_niso_lab2 -question 1 -clause "0.5 2 1 -3 -4 0" -assignment 0000
1
[pkl@phi ocamlec]$ app_niso_lab2 -question 1 -clause "0.5 2 1 -3 -4 0" -assignment 0011
0
```

Exercise 2. *(10 % of the marks) Implement a routine for importing MAXSAT instances on the WDIMACS file format. A WDIMACS file is a text file where each line either*

- *begins with the letter* `c`*, indiciating that the rest of the line is a comment*
- *begins with* `p cnf n m x` *where* `n` *is the number of variables,* `m` *is the number of clauses, and* `x` *is an optional parameter which may or may not be present, or*
- *describes a clause, as explained in Exercise 1.*

To test that the import routine works correctly, you should count the number of clauses which are satisfied by a given assignment.

Input arguments:

- `-wdimacs` *name of file on WDIMACS format*
- `-assignment` *an assignment as a bitstring*

Output:

- *the number of clauses in the file which are satisfied by the assignment*

Example:

```
[pkl@phi ocamlec]$ cat example.wcnf
c Example file
p wcnf 4 2
0 1 2 3 4 0
0 -1 -2 3 -4 0
[pkl@phi ocamlec]$ app_niso_lab2 -question 2 -wdimacs example.wcnf -assignment 0000
1
[pkl@phi ocamlec]$ app_niso_lab2 -question 2 -wdimacs example.wcnf -assignment 0001
2
```

Exercise 3. *(40 % of the marks)*

Design an evolutionary algorithm for the MAXSAT problem. You can use any of the techniques described in the module so far, or other approaches. Your algorithm should return a solution as an assignment within a specified time budget (in seconds). The population should be randomly initialised before each repetition.

Your solution will be evaluated on instances from the MAXSAT Evaluation 2017 competition. You can download instances from `http://mse17.cs.helsinki.fi` *(complete unweighted benchmarks).*

Input arguments:

- `-wdimacs` *name of file on WDIMACS format*
- `-time_budget` *number of seconds per repetition*
- `-repetitions` *the number of repetitions of the algorithm*

Output:

The output should be one line per repetition, where the following values are shown separated by the tab character

```
t nsat  xbest
```

where

- `t` *is the runtime (number of generations times population size)*
- `nsat` *is the number of satisfied clauses in the returned solution*
- `xbest` *the solution found by the algorithm*

Example:

```
[pkl@phi ocamlec]$ app_niso_lab2 -question 3 \
       -wdimacs example.wcnf -time_budget 1 -repetitions 5
1700300 2 1110
1755100 2 0010
1705500 2 1000
1734100 2 0111
1743200 2 0111
```

Exercise 4. *(10 % of the marks)*

Describe your algorithm from Exercise 3 in the form of pseudo-code (see Lab 1 for an example). The pseudo-code should be sufficiently detailed to allow an exact re-implementation.

Exercise 5. *(30 % of the marks)*

Your algorithm is likely to have several parameters, such as the population size, that must be set before running the algorithm. Choose three parameters which you think are essential for the behaviour of your algorithm. Run a set of experiments to determine the impact of these parameters on the quality (number of satisfied clauses) of the solutions obtained. For each parameter setting, run the algorithm on some selected problem instances from the MAXSAT competition (MSE17 complete unweighted benchmarks) which you can download from the following URL:

`http://mse17.cs.helsinki.fi/benchmarks.html`

Note that some of the instances in the MAXSAT competition are very large. Unless you have a significant computational resources available, you might find it useful to start with the smallest instances. You will also need to set a reasonable time out value that allows you to run a sufficient number of experiments.

For each parameter setting and problem instance, make 100 repetitions. Plot the results as boxplots, and discuss your findings. Clearly indicate which of the instances you tested your code on.

A. Docker Howto

Follow these steps exactly to build, test, save, and submit your Docker image. Please replace *abc123* in the text below with your username.

1. Install Docker CE on your machine from the following website:
   `https://www.docker.com/community-edition`
2. Copy the PDF file from Exercises 4 and 5 all required source files, and/or bytecode to an empty directory named niso2-abc123 (where you replace abc123 with your username).

```
mkdir niso2-abc123
cd niso2-abc123/
cp ../exercise.pdf .
cp ../abc123.py .
```

3. Create a text file `Dockerimage` file in the same directory, following the instructions below.

```
# Do not change the following line. It specifies the base image which
# will be downloaded when you build your image. We will release a new
# base image for each lab.

FROM pklehre/niso-lab2

# Add all the files you need for your submission into the Docker image,
# e.g. source code, Java bytecode, etc. In this example, we assume your
# program is the Python code in the file abc123.py. For simplicity, we
# copy the file to the /bin directory in the Docker image. You can add
# multiple files if needed.

ADD abc123.py /bin

# Install all the software required to run your code. The Docker image
# is derived from the Debian Linux distribution. You therefore need to
# use the apt-get package manager to install software. You can install
# e.g. java, python, ghc or whatever you need. You can also
# compile your code if needed.

# Note that Java and Python are already installed in the base image.

# RUN apt-get update
# RUN apt-get -y install python-numpy

# The final line specifies your username and how to start your program.
# Replace abc123 with your real username and python /bin/abc123.py
# with what is required to start your program.

CMD ["-username", "abc123", "-submission", "python /bin/abc123.py"]
```

4. Build the Docker image as shown below. The base image `pklehre/niso-lab1` will be downloaded from Docker Hub

```
docker build . -t niso2-abc123
```

5. Run the docker image to test that your program starts. A battery of test cases will be executed to check your solution.

```
docker run niso2-abc123
```

6. Once you are happy with your solution, compress the directory containing the Dockerfile as a zip-file. The directory should contain the source code, the Dockerfile, and the PDF file for Exercise 4 and 5. The name of the zip-file should be `niso2-abc123.zip` (again, replace the abc123 with your username).

7. Submit the zip file `niso2-abc123.zip` on Canvas.