LINFO1361: Artificial Intelligence
Assignment 4: Local Search and Propositional Logic

Nicolas Golenvaux, Yves Deville

April 2022

Guidelines

- This assignment is due on **Thursday 13 May, 6:00 pm**.
- *No delay* will be tolerated.
- *Document* your source code (at least the difficult or more technical parts of your programs). Python docstrings for important classes, methods and functions are also welcome.
- Copying code or answers from other groups (or from the internet) is strictly forbidden. Each source of inspiration must be clearly indicated.
- Source code shall be submitted on the online *INGInious* system. Only programs submitted via this procedure will be graded. No program sent by email will be accepted.
- Respect carefully the *specifications* given for your program (arguments, input/output format, etc.) as the program testing system is *fully automated*.
- The answer to questions must be given by filling in the latex template provided. The final file must be submitted on *gradescope*. No report sent by email will be accepted.
- Nothing must be modified in the template except your answer that you insert in the *answer* environments as well as your names and your group number. The names are provided through the command *students* while the command *group* is used for the group number. The dimensions of *answer* fields *must not* be modified either. For the tables, put your answer between the "&" symbols. Any other changes to the file will *invalidate* your submission.
- To submit on gradescope, go to `https://gradescope.com` and click on the "log in" button. Then choose the "school credentials" option and search for *UCLouvain Username*. Log in with your global username and password. Find the course LINFO1361 and the Assignment 4, then submit your report. Only one member should submit the report and add the second as group member.
- For those who have not been automatically added to the course, at the right bottom of gradescope homepage, ckick on "Enroll on course" button and type the code *86KJVB*.
- Check this link if you have any trouble with group submission `https://help.gradescope.com/article/m5qz2xsnjy-student-add-group-members`

Deliverables

- The answers to all the questions in a report **on INGInious. Do not forget to put your group number on the front page as well as the INGInious id of both**

group members.

- The following files are to be submitted:
  - `vertexcover_max.py`: your *maxvalue* local search for the Maximum Vertex Cover problem, to submit on INGInious in the *Assignment4: Cover Vertex max value* task.
  - `vertexcover_random_max.py`: your *randomized max value* local search for the Maximum Vertex Cover problem, to submit on INGInious in the *Assignment4: Vertex Cover randomized maxvalue* task.
  - `queen_solver.py`: which contains `get_expression` method to solve the Queen problem, to submit on INGInious in the *Assignment4: Queen Problem* task.

1 The Vertex Cover problem (13 pts)

In this assignment you will design an algorithm to solve the vertex cover problem. You are provided with an integer k and an undirected graph. Your task is to find the set of k vertices that *touch* the maximal number of edges. Figure 1 shows an example of a maximal solution for vertex cover problem.

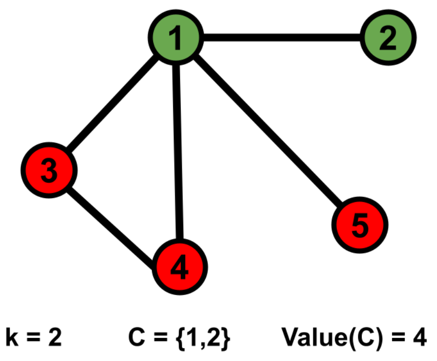

Figure 1: Example of a vertex cover problem.

Formally, the cover vertex problem is defined as follows: given a graph $G = (V, E)$ where V is the set of vertices and E the set of undirected edges $e = (u, v)$ linking the vertices u and v, find a set C of k vertices that covers a maximal number of edges. An edge $e = (u, v) \in E$ is said to be *covered* if at least one of its endpoint u or v is in C.

1.1 Objective function

A solution C to vertex cover problem is evaluated by counting the number of edges covered.

$$Value(C) = |E_C| \tag{1}$$

where

$$E_C = \{e = (u, v) \mid u \text{ and/or } v \in C\} \qquad \text{AND} \qquad |C| = k \tag{2}$$

1.2 Neighborhood

In local search, we have to build at each decision time a neighborhood in which a solution is picked depending on a specific criterion. We suggest to only consider local solutions with exactly k vertices, and to use a simple swap action to build the neighborhood. This action consists of the swap of two vertices u and v where $u \in C$ and $v \notin C$. This means that using this type action, each solution has $k * (m - k)$ neighbors where $m = |V|$ is the number of vertices in the graph G.

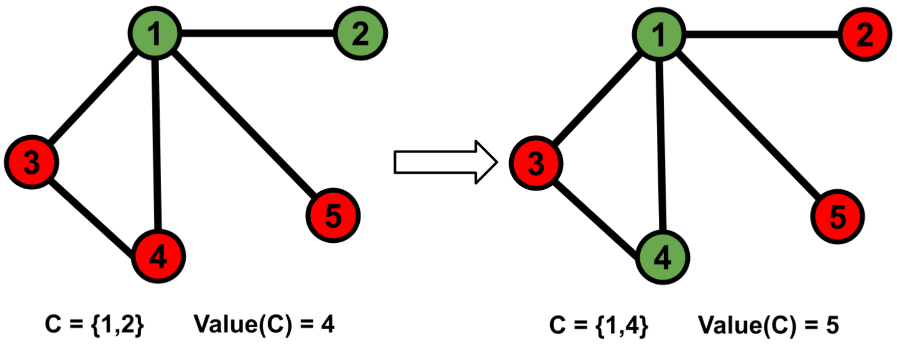

Figure 2: Swap of two vertices

You are not obliged to use the proposed objective function as well as techniques for building the neighborhood. You can use your owns but you have to specify in your report the ones you used. However, the capacity constraint of the vertex cover cannot be violated during building of the neighborhood. Moreover, note that your result will be evaluated with the function described in 1 and 2.

A vertex cover input contains $n+1$ lines. The first line contains 3 integers which represent the size k of the cover, the number of vertices $m = |V|$ and the number of edges $n = |E|$ of the graph $G = (V, E)$. The n following lines present each edge $e \in E$. Each line contains three integers. The first one is the edge's index and the second and third ones indicate the indices of the two vertices u and v linked by that edge.

The format for describing the different instances on which you will have to test your program is the following:

Input format

$$\begin{array}{lll} k & m & n \\ 1 & u_1 & v_1 \\ 2 & u_2 & v_2 \\ \dots & & \\ n & u_n & v_n \end{array}$$

For this assignment, you will use *Local Search* to find good solutions to the vertex cover problem. The test instances can be found on Moodle. A template for your code is also provided. The output format **must** be the following:

Output format

$c_1\ c_2\ \dots\ c_k$

Where c_1 c_2 ... c_k are the indices of the vertices in the cover set C. There is no specific order required in the output of the vertices indices.
For example, the input and an output for the vertex cover problem of Figure 1 could be:

Input format

```
2 5 5
1 1 2
2 1 3
3 1 4
4 3 4
5 1 5
```

Output format

```
1 4
```

Diversification versus Intensification

The two key principles of Local Search are intensification and diversification. Intensification is targeted at enhancing the current solution and diversification tries to avoid the search from falling into local optima. Good local search algorithms have a tradeoff between these two principles. For this part of the assignment, you will have to experiment this tradeoff.

Questions

1. **(1 pt)** Formulate the vertex cover problem as a Local Search problem (problem, cost function, feasible solutions, optimal solutions).
2. **(5 pts)** You are given a template on Moodle: *vertexcover.py*. Implement your own extension of the *Problem* class from `aima-python3`. Implement the `maxvalue` and `randomized maxvalue` strategies. To do so, you can get inspiration from the *randomwalk* function in *search.py*. Your program will be evaluated on 15 instances (during 1 minute) of which 5 are hidden. We expect you to solve at least 12 out the 15.
   (a) `maxvalue` chooses the best node (i.e., the node with maximum value) in the neighborhood, even if it degrades the quality of the current solution. The `maxvalue` strategy should be defined in a function called *maxvalue* with the following signature: `maxvalue(problem,limit=100,callback=None)`.
   (b) `randomized maxvalue` chooses the next node randomly among the 5 best neighbors (again, even if it degrades the quality of the current solution). The `randomized maxvalue` strategy should be defined in a function called *randomized_maxvalue* with the following signature:
   `randomized_maxvalue(problem,limit=100,callback=None)`.
3. **(3 pts)** Compare the 2 strategies implemented in the previous question and `randomwalk` defined in *search.py* on the given vertex cover instances. Report, in a table, the computation time, the value of the best solution and the number of steps needed to reach the best result. For the randomized max value and the random walk, each instance should be tested 10 times to reduce the effects

of the randomness on the result. When multiple runs of the same instance are executed, report the mean of the quantities.

4. **(4 pts)** Answer the following questions:
   (a) What is the best strategy?
   (b) Why do you think the best strategy beats the other ones? What conclusions can be drawn on the vertex cover problem ?
   (c) What are the limitations of each strategy in terms of diversification and intensification?
   (d) What is the behavior of the different techniques when they fall in a local optimum?

2 Propositional Logic (7 pts)

2.1 Models and Logical Connectives (1 pt)

Consider the vocabulary with four propositions A, B, C and D and the following sentences:

- $\neg(A \land B) \lor (\neg B \land C)$
- $(\neg A \lor B) \Rightarrow C$
- $(A \lor \neg B) \land (\neg B \Rightarrow \neg C) \land \neg(D \Rightarrow \neg A)$

Questions

1. **(1 pt)** For each sentence, give its number of valid interpretations, i.e. the number of times the sentence is true (considering for each sentence **all the proposition variables** A, B, C and D).

2.2 N-Queens Problem (6 pts)

The N-Queens Problem can be defined as follow. Given a chessboard of shape $n\_rows \times n\_columns$, you have to place n_queens queens on the chessboard such that no queen can attack any other in a single movement. (A queen can move on any cell in the same row, column or diagonal.) A N-Queens problem uses n_queens queens such that $n\_rows = n\_columns = n\_queens$. The N-Queens Problem asks whether such a placement exists. Figures 3 below shows an example of a valid solution (left) and an invalid solution (right).

Your task is to model this problem with propositional logic. We define $n\_rows \times n\_columns$ variables: C_{ij} is *true* iff there is a queen on cell at position (i, j); *false* otherwise. The chessboard origin $(0, 0)$ is at left top corner.

Questions

1. **(2 pts)** Explain how you can express this problem with propositional logic. For each sentence, give its meaning.
2. **(2 pts)** Translate your model into Conjunctive Normal Form (CNF).

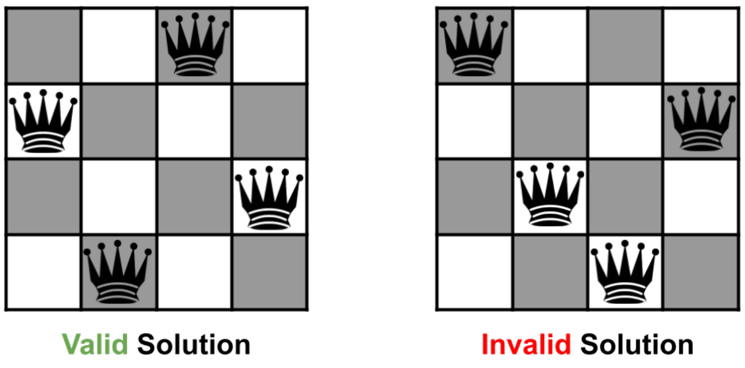

Figure 3: Example of 4-Queens Solutions

On the Moodle site, you will find an archive `queen.zip` containing the following files:

- `instances` is a directory containing a few instances for you to test this problem.
- `solve_linux.py / solve_mac.py` is a python file used to solve the n-queens Problem, whether you machine is running on Linux or Mac.
- `queen_solver.py` is the skeleton of a Python module to formulate the problem into an CNF.
- `minisat.py` is a simple Python module to interact with MiniSat.
- `clause.py` is a simple Python module to represent your clauses.
- `minisatLinux / minisatMac` is a pre-compiled MiniSat binary that should run on the machines in the computer labs or your machine depending on the os.

To solve the n-queens Problem for instance on Linux machine, enter the following command in a terminal:

```
python3 solve_linux.py INSTANCE_FILE
```

where `INSTANCE_FILE` is the n-queens instance file. To have different versions of a problem of specific size, the instance file is not only composed of the size of the problem but some queens are already placed on a fixed cell. It is described as follows:

Instance input format

$$\begin{array}{ll} n & k \\ q_1.i & q_1.j \\ q_2.i & q_2.j \\ \dots & \\ q_k.i & q_k.j \end{array}$$

where n represents the value of the three (03) parameters of the problem shape since the chessboard is square. k is the number of queen's information provided. The k following lines represent information about the fixed queens and are composed of two integers representing respectively values of x axis and y axis, all starting from 0.

Example of input

```
8 3
1 5
7 1
3 2
```

Questions

3. **(2 pts)** Modify the function `get_expression(size)` in `queen_solver.py` such that it outputs a list of clauses modeling the n-queens problem for the given input. Submit your code on INGInious inside the *Assignment4: N-Queens Problem* task. The file `queen_solver.py` is the *only* file that you need to modify to solve this problem. Your program will be evaluated on 10 instances of which 5 are hidden. We expect you to solve at least 8 out the 10.