LINGI2261: Artificial Intelligence
Assignment 4: Local Search and Propositional Logic

François Aubry, Gael Aglin, Yves Deville
November 2018

Guidelines

- This assignment is due on **Wednesday 12 December, 6:00 pm**.
- *No delay* will be tolerated.
- Not making a *running implementation* in *Python 3* able to solve (some instances of) the problem is equivalent to fail. Writing some lines of code is easy but writing a correct program is much more difficult.
- *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.
- Indicate clearly in your report if you have *bugs* or problems in your program. The online submission system will discover them anyway.
- Copying code or answers from other groups (or from the internet) is strictly forbidden. Each source of inspiration must be clearly indicated. The consequences of *plagiarism* is *0/20 for all assignments*.
- Source code shall be submitted on the online *INGInious* system. Only programs submitted via this procedure will be graded. No report or 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*.

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 on *INGInious* inside the *Assignment 4* task(s):
  - `knapsack_maxvalue.py`: your *maxvalue* local search for Knapsack problem, to submit on INGInious in the *Assignment4: Knapsack problem : maxvalue* task.
  - `knapsack_randomized_maxvalue.py`: your *randomized maxvalue* local search for Knapsack problem, to submit on INGInious in the *Assignment4: Knapsack randomized maxvalue* task.
  - `gc_sol.py`: which contains `get_clauses` method to solve the Graph Coloring problem, to submit on INGInious in the *Assignment4: Graph Coloring Problem* task.

1 The Knapsack Problem (13 pts)

In this assignment you will design an algorithm to solve the infamous Knapsack Problem. You are provided with a knapsack with limited space (capacity) and a collection of items with different utilities and weights. Your task is to maximize the utility of items packed into your knapsack without exceeding its total capacity. Figure 1 shows an example of a small knapsack problem.

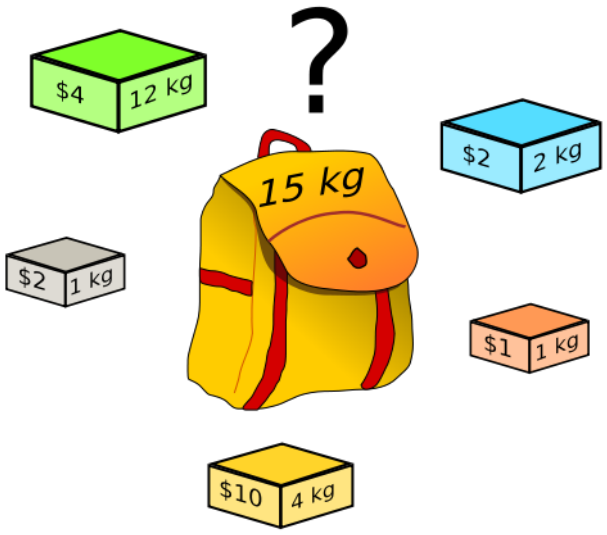

Figure 1: Example of a knapsack problem: which boxes should be chosen to maximize the amount of money while still keeping the overall weight under or equal to 15 kg?
Source: https://wikipedia.org

Our knapsack problem has a particularity. Items are from different types and for this reason, we have to ensure that items we will pick into the knapsack are not in conflicts each other. For example, it is not relevant to pick up a rabbit and a carrot. You realize that the total utility will not be the same when the carrot will be in the rabbit's stomach. Fortunately, the list of items in conflicts is provided. So, in addition to ensure that the items you packed in the knapsack do not exceed the knapsack capacity, you must ensure that there is no conflict.

A knapsack input contains $n+2$ lines. The first line contains an integer which represent the number of items in the problem, n. The n following lines present the data for each of the items. Each line, $i \in 1 \ldots n$ contains several integers. The first one is item's index, the second is its weight w_i, the third is its utility u_i followed by the list c (items indexes separated by space) of items in conflict with it. The list of items in conflict can be empty. The last line contains a number K, the capacity of the knapsack.

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

Input format

n
1 w_1 u_1 c_1 c_2
2 w_2 u_2
...
n w_n u_n c_1
K

For this assignment, you will use *Local Search* to find good solutions to the Knapsack 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

$weight : W$ $utility : U$
$Items : [p_1, p_2, \ldots, p_m]$
$Capacity : K$
$STEP : S$

Where W is the total weight of the solution and U its utility. p is the list of m packed items and S is the number of steps before reaching the solution. **The items indexes start from 1. Pay attention to respect it in the output printing format.**

For example, the input and output for our knapsack problem could be:

Input format

6
1 200 900 3
2 100 25
3 400 1200 1 5
4 50 40
5 50 100 3
6 150 25
600

Output format

$weight : 550$ $utility : 1265$
$Items : [3, 2, 4]$
$Capacity : 600$
$STEP : 3$

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 pts)** Formulate the Knapsack problem as a Local Search problem (problem, cost function, feasible solutions, optimal solutions).
2. **(5 pts)** You are given a template on Moodle: *knapsack.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*.
   (a) `maxvalue` chooses the best node (i.e., the node with minimum 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)*.

   Describe in your report how you construct your initial solution and how your successor function works.
3. **(4 pts)** Compare the 2 strategies implemented in the previous question between each other and with `randomwalk`, defined in *search.py* on the given knapsack instances. Report, in a table, the computation time, the value of the best solution and the number of steps when the best result was reached. For the second and third strategies, each instance should be tested 10 times to eliminate the effects of the randomness on the result. When multiple runs of the same instance are executed, report the means of the quantities.
4. **(3 pts)** Answer the following questions:
   (a) What is the best strategy?
   (b) Why do you think the best strategy beats the other ones?
   (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 pts)

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

- $(\neg A \vee C) \wedge (\neg B \vee C)$
- $(C \Rightarrow \neg A) \wedge \neg(B \vee C)$
- $(\neg A \vee B) \wedge \neg(B \Rightarrow \neg C) \wedge \neg(\neg D \Rightarrow A)$

Questions

1. **(1 pts)** 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 Graph Coloring Problem (6 pts)

The Graph Coloring Problem can be defined as follow. Given an undirected graph $G = (V, E)$, a graph coloring assigns a color to each node, such that all adjacent nodes have a different color. A graph coloring using at most k colors is called a k-coloring. The Graph Coloring Problem asks whether a k-coloring for G exists. Figures 2 below shows an example of a valid coloring (left) and an invalid coloring (right).

Figure 2: Example of graph coloring

Your task is to model this problem with propositional logic. We define $|V| \times k$ variables: $X_{vc} = 1$ iff node v is assigned color c; 0 otherwise.

Questions

1. **(2 pts)** Explain how you can express this problem with propositional logic. What are the relations and how do you translate them?
2. **(2 pts)** Translate your model into Conjunctive Normal Form (CNF) and write it in your report.

On the Moodle site, you will find a zip `cg.zip` containing the following files:

- `given_instances` are a few graph instances for you to test this problem.
- `graph.py` is a Python module to load and manipulate the graph instances described above.

- `solve.py` is a python file used to solve the Graph Coloring Problem.
- `gc_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` is a pre-compiled MiniSat binary that should run on the machines in the computer labs and your machine if you have Linux.

To solve the Graph Coloring Problem, enter the following command in a terminal:

```
python3 solve.py GRAPH_INSTANCE NB_COLORS
```

where `GRAPH_INSTANCE` is the graph instance file and `NB_COLORS` is number of colors.

For instance,

```
python3 solve.py instances/square.col 3
```

will try to find a 3-coloring of the graph represented in `square.col`.

Questions

3. **(2 pts)** Modify the function `get_clauses(G, nb_colors)` in `gc_solver.py` such that it outputs a list of clauses modeling the graph coloring problem for the given graph and number of colors. Submit your functions on INGInious inside the *Assignment4: Graph Coloring Problem* task.

   The file `gc_solver.py` is the *only* file that you need to modify to solve this problem.