Combinatorial Optimisation

Computer Assignment 2

Important information: read carefully!

Requirements

- An understanding of Lagrangian Relaxation and Subgradient Optimisation.
- This assignment requires programming in Python (Python 3).

Hand-in information

- The hand-in deadline is Sunday, October 13 at 23:59.
- The assignment can be made in groups of up to two students.
- We use an online web form and Codegrade (in Canvas).
- Two files should be submitted through Canvas. One single python code file (**SubmissionFunctions.py**) and a json file (**Theory.json**) from the online web.
- Please upload the two files directly, without placing them in a folder.
- See the Canvas assignment for more details.

Cheating and plagiarism

- You are not allowed to share code except with your group member, but are allowed to discuss the assignment and related theory with fellow students. Code will be checked for plagiarism.
- Any attempt to mislead or alter the automated submission and grading system or to prevent correct grading will be considered as cheating.
- Cheating/plagiarism is not tolerated and will be reported to the Examination Board.

Contact in case of issues

- For issues regarding Codegrade or errors in the assignment contact (j.cho@ese.eur.nl) as soon as possible. This includes issues with the hand-in, missing the deadline, and (suspected) incorrect grading. Before sending an email, please check one more time ensuring that if the file names are correct (especially capital letters), and that your group members are listed correctly. When you send an email, please add [Combinatorial Optimisation 2024] to the subject line.
- Other questions should be asked to the lecturer or your teaching assistant, as usual.

Additional instructions for your code

- Your code should be valid Python 3.
- Your code should include (brief) comments such that it is clear what each part does.
- A single file (**SubmissionFunctions.py**) should contain all the functions.
- Do not include any other functions or extra codes to SubmissionFuctions.py file. (e.g. codes for plotting and testing)

Codegrade scores for theory questions

- Check the Codegrade output by clicking on your submission score.
- Double-check that the file name is correct (**Theory.json**).
- For the theory questions: your score is either 0% or 100% until the official grading.
- A score of 100% indicates that your answers are parsed correctly. After the deadline, your answers will be graded.
- A score of 0% indicates that your answers cannot be parsed correctly. If the issue cannot be resolved, contact the staff member responsible for grading the assignment.

Codegrade scores for code

- Check the Codegrade output by clicking on your submission score.
- A score of 100% indicates that your code generates results that are close to those of the reference codes within some tolerances.
- The failed instances indicate that your code has an issue or that the results are significantly different from the expected answers.

Exercise 1 Deriving lower and upper bounds for the set covering problem by using Lagrangian relaxation

Consider the set covering problem (for more details we refer to pages 17–19 of the slides on Formulations). The problem can be formulated as

$$\begin{aligned} z^* = \min \quad & \sum_{j=1}^{n} c_j x_j & & \\ \text{s.t.} \quad & \sum_{j=1}^{n} a_{ij} x_j \geq 1 & & \forall\, i \in \{1, \ldots, m\}, \\ & x_j \in \mathbb{B} & & \forall\, j \in \{1, \ldots, n\}. \end{aligned} \tag{1}$$

As follows from the formulation, a problem instance is given by

- the dimensions $n, m \in \mathbb{N}_{\geq 1}$,
- a strictly positive cost vector $\mathbf{c} \in \mathbb{R}^n_{>0}$,
- an adjacency matrix $A \in \mathbb{B}^{m \times n}$ with elements a_{ij} for $i \in \{1, \ldots, m\}$, $j \in \{1, \ldots, n\}$.

Furthermore, a solution is given by the vector $\mathbf{x} \in \mathbb{B}^n$ and the optimal objective value is $z^* \in \mathbb{R}$.

We will apply Lagrangian relaxation by dualizing the constraints (1). In this assignment we will solve the Lagrangian dual problem by subgradient optimisation resulting in a lower bound on the optimal objective value z^*. Moreover, we will derive an upper bound on z^* by using a Lagrangian heuristic.

In your implementation, keep the following points in mind:

- Write all functions in a single file.
- Do not use nested functions, anonymous functions, or subfunctions.
- Call functions written for other parts of the exercise when appropriate.
- The data types of all input or output are specified in the base code.
- Use `NumPy` as follows: `import numpy as np`.
- Use the instances stored in (`small_instance_A.text`, `small_instance_c.text`) and `large_instance_A.text`, `large_instance_c.text`) found on Canvas to call, test, and debug your functions.
- Read input files using `np.loadtxt` in `NumPy` with the `ndmin` parameter set to 2; e.g.,
  `A = np.loadtxt('small_instance_A.txt', ndmin=2)`, or
  `c = np.loadtxt('small_instance_c.txt', ndmin=2)`.
  This enables you to perform matrix calculations directly on `NumPy` arrays.
- All the vectors, both input and output, should be treated as two-dimensional `numpy.ndarray` objects. A one-dimensional vector is regarded as a column vector.
  For example, `c = np.array([[3], [3], [3], [5], [4]])`.
- Scalars must *not* be contained within an array, e.g., `obj_lagrange` and `obj_feas`.
- In Python 3, `lambda` is a reserved keyword used to create anonymous functions. Instead, `labda` is used in the base code. Be cautious with the names of the variables.

a. Create a function

```
Lagrangian(c, A, labda) -> (obj_lagrange, x_lagrange)
```

with

`c`	:	cost vector **c** of the instance,
`A`	:	adjacency matrix A of the instance,
`labda`	:	vector of Lagrangian multipliers,
`x_lagrange`	:	optimal solution of the Lagrangian relaxation for `labda`, refer to Exercise 4a in Set 3 with any ties set to zero,
`obj_lagrange`	:	optimal objective value of the Lagrangian relaxation for `labda`.

Note: as specified above, of all optimal solutions of the Lagrangian relaxation for `labda` you must construct the optimal solution `x_lagrange` with as many elements set to zero.

b. Create a function

```
InfeasToFeas(c, A, x_infeas) -> (obj_feas, x_feas)
```

with

`c`	:	cost vector **c** of the instance,
`A`	:	adjacency matrix A of the instance,
`x_infeas`	:	potentially infeasible primal solution,
`x_feas`	:	feasible primal solution constructed by applying the Greedy heuristic of Exercise 4c in Set 3 to `x_infeas` where the smallest appropriate index is chosen in case of ties,
`obj_feas`	:	primal objective value corresponding to `x_feas`.

Note: as specified above, if during the Greedy heuristic there are multiple choices, choose from these the one with the smallest index.

c. Create a function

```
UpdateLabda(A, labda, LB, UB, x_lagrange, rho) -> labda_next
```

with

`A`	:	adjacency matrix A of the instance,
`labda`	:	vector of Lagrangian multipliers,
`LB`	:	lower bound on the optimal primal objective value z^*,
`UB`	:	upper bound on the optimal primal objective value z^*,
`x_lagrange`	:	optimal solution of the Lagrangian relaxation for `labda`,
`rho`	:	scaling factor ρ to be used in the subgradient step, refer to pages 36–38 of the slides on Relaxations and Exercise 7 in Set 3,
`labda_next`	:	new vector of Lagrangian multipliers constructed by applying the subgradient step, refer to pages 36–38 of the slides on Relaxations and Exercise 7 in Set 3.

d. Create a function

```
SubgradientOpt(c, A, labda_init, rho_init, k)
-> (LB_best, UB_best, x_best, LB_list, UB_list)
```

with

`c`	:	cost vector $\mathbf{c}$ of the instance,
`A`	:	adjacency matrix A of the instance,
`labda_init`	:	initial vector of Lagrangian multipliers to be used,
`rho_init`	:	initial scaling factor ρ to be used in the subgradient step,
`k`	:	(strictly positive) number of subgradient iterations to perform,
`LB_best`	:	best found lower bound on the optimal primal objective value z^*,
`UB_best`	:	best found upper bound on the optimal primal objective value z^*,
`x_best`	:	best found feasible primal solution,
`LB_list`	:	vector of found lower bounds on the optimal primal objective value z^* for each of the `k` iterations,
`UB_list`	:	vector of found upper bounds on the optimal primal objective value z^* for each of the `k` iterations.

The `SubgradientOpt` function should use the functions in Parts **a**, **b**, and **c**. So do not replicate code. In particular, in a single iteration first a new lower and upper bound is computed, and then the vector of Lagrangian multipliers is updated. Furthermore, in each iteration `UpdateLabda` of Part **c** should be called with

`labda`	:	current vector of Lagrangian multipliers,
`LB`	:	*current*(!) lower bound on the optimal primal objective value z^*,
`UB`	:	*best*(!) upper bound on the optimal primal objective value z^*,
`x_lagrange`	:	current optimal solution of the Lagrangian relaxation for `labda`,
`rho`	:	`rho_init` divided by the current iteration number, where the iteration number runs from 1 up to and including `k`.

e. Write a function that shows the graph of both the *current* and the *best* lower and upper bounds found in each iteration. (You do not hand in this function.)

f. Derive a lower and upper bound for the small problem instance in `small_instance.mat`, by applying the subgradient optimisation of Part **d** with `labda_init` $= (1, \ldots, 1)$, `rho_init` $= \frac{1}{2}$, and `k` $= 50$. Plot the corresponding graph using Part **e**. Observe and interpret the results. (You do not hand in these results, use it for Part **h**.)

g. Derive a lower and upper bound for the large problem instance in `large_instance.mat`, by applying the subgradient optimisation of Part **d** with `labda_init` $= (1, \ldots, 1)$, `rho_init` $= 200$, and `k` $= 1000$. Plot the corresponding graph using Part **e**. Observe and interpret the results. (You do not hand in these results, use it for Part **h**.)

h. Use the previous parts to answer the multiple-choice questions in the online web-form.

