Algorithm Engineering **Summer 2026**

Dr. Phillip Keldenich
Dr. Dominik Krupke

Submission due: 21.05.2026 13:15

Homework 3

Work in teams of two. Homework is due every two weeks.

Submission format: Host your solution in a private Git repository on one of the following platforms and add Rouven as a collaborator: IBR GitLab (`kniep`), TU GitLab (`r.kniep`), GitHub (`rkniep`), or Codeberg (`rkniep`). If you want to use a different platform, please ask first.

Before the deadline, send the link to your submission commit by e-mail to `kniep@ibr.cs.tu-bs.de`.

Exercise 1 (Branch and Cut): **(3+5+5+7 points)**

The `exercise01` subdirectory for this exercise contains a skeleton Branch & Bound implementation in Python for the Knapsack problem with mutual exclusivity constraints. As in the Knapsack problem, the input for the problem consists of a set of items $X = \{x_1, \ldots, x_n\}$ with profits $p_i \in \mathbb{R}_{\geq 0}$ and weights $w_i \in \mathbb{R}_{\geq 0}$, and a capacity limit $c \in \mathbb{R}$. The goal is to identify a solution set $S \subseteq X$ maximizing the profit $P(S) = \sum_{x_i \in S} p_i$ that has weight $W(S) = \sum_{x_i \in S} w_i \leq c$. Additionally, each item x_i has an associated set of *excluded items* $E_i \subseteq X$. If a solution contains x_i, it must not contain any item $x_j \in E_i$; this relationship is symmetric, i.e., we have $x_i \in E_j$ if and only if $x_j \in E_i$.

This problem can also be seen as a version of independent set on graphs with vertex weights and profits, where the goal is to select an independent set that is as profitable as possible while staying below a weight limit.

The Branch and Bound implementation can be found in

`exercise01/src/mutually_exclusive_knapsack/branch_and_bound.py`.

It is part of a package that can be installed in a virtual environment by running `pip install -e .` in the `exercise01` directory. Doing so installs a commandline entrypoint named `mek-solve`, which can be passed an instance of the problem in JSON format. The repository also contains a benchmark instance file named `benchmark_150.json` with 150 items, which we want to use for benchmarking our implementation.

Under the hood, the implementation uses the linear relaxation of the following straightforward formulation of the problem to compute bounds via the open-source solver HiGHS.

$$\max \sum_{i=1}^{n} p_i x_i \text{ subject to} \tag{1}$$

$$\sum_{i=1}^{n} w_i x_i \leq c \tag{2}$$

$$\forall i \; \forall j \in E_i : x_i + x_j \leq 1 \tag{3}$$

$$\forall i : x_i \in \{0, 1\}. \tag{4}$$

Here, constraint 2 enforces that the solution stays below the weight limit, constraint 3 enforces that two mutually exclusive items cannot both be selected, and constraint 4 ensures that each item is either fully taken ($x_i = 1$) or not taken at all ($x_i = 0$).

While we will explore linear programming later in the lecture, for now you should not have to interface with the LP solver directly; the code needed for setting up the model and calling the solver is already in place.

While the implementation should produce correct results as is, it lacks several components that will improve its performance dramatically. Perform the following fixes in order; report the runtime and number of nodes visited reported by `mek-solve benchmark_150.json` after each step.

a) The implementation does not perform constraint propagation at all; the method stub `constraint_propagation` does nothing. Convince yourself that the linear relaxation automatically ensures that fixing $x_i = 1$ results in $x_j = 0$ for all $x_j \in E_i$. Also, convince yourself that the linear relaxation may set some $x_j > 0$ even if w_j is greater than the remaining capacity $c - \sum_{x_i \in F} w_i$, where F contains the items x_i that have been fixed to $x_i = 1$. Implement the following constraint propagation, which is necessary to avoid needlessly visiting branch and bound nodes: set $x_j = 0$ for all items for which w_j is larger than the remaining capacity, after the items x_i that have already been fixed to 1 in the given partial assignment are considered.

b) The existing lower-bound heuristic is very weak; it just takes the single most profitable item with $w_i \leq c$. Implement a better heuristic and use it to initialize the lower bound. You may, for instance, order the items by non-increasing efficiency p_i/w_i and then greedily construct a solution; in this case, you should consider multiple starting points in the greedy ordering; try to beat a value of 1934 for the benchmark instance.

c) Recall that the fundamental idea of Branch & Cut is to improve the bound produced by the linear relaxation by the addition of *cutting planes*, which are additional linear constraints. These constraints have two important properties: Firstly, they are satisfied by all integral solutions; after all, we do not want them to accidentally prevent us from finding optimal solutions. Secondly, they are violated by the solution to the current linear relaxation; otherwise, they would not change the solution and thus could not improve the bound. In practice, one usually generates these violated linear constraints heuristically, knowing the full (mixed) integer program and given the relaxed solution; there are a several families or templates of cutting planes that are integrated in all serious MIP solvers.

   For our problem, there are two important families of cutting planes that may be useful, namely *clique cuts* and the *Knapsack cover* cuts introduced in the lecture. In many cases, the linear relaxation can be improved by these cuts. Finding a violated cut in these cases be done by simple greedy heuristics; if they fail, we can simply move on by branching.

   Firstly, implement the Knapsack cover cuts[1]: given a fractional solution $x^* \in [0, 1]^n$, try to find a subset D of the indices $\{1, \ldots, n\}$ such that $\sum_{i \in D} w_i > c$, i.e., the items in

[1]The method stub is called `knapsack_cover_cut_separation`.

D do not fit into the container, but $\sum_{i \in D} x_i^* > |D| - 1$. To do this, you can try sorting the items both by non-increasing x_i^* or by non-decreasing $(1 - x_i^*)/w_i$, constructing D greedily, and stopping whenever the total weight exceeds the container capacity c.

Make sure you only generate cuts that are violated by at least some threshold value $\varepsilon > 0$ to avoid numerical issues and bad cuts; the implementation defines a corresponding global constant.

d) Clique cuts are cuts that make use of the following observation. Whenever we have a subset C of indices of mutually exclusive items, i.e., $x_i \in E_j$ for each $i, j \in C, i \neq j$, at most one of the items in C can be selected. In terms of a linear constraint, we can be sure that $\sum_{i \in C} x_i \leq 1$ will be satisfied by all integral solutions. However, even for only three mutually exclusive items $C = \{1, 2, 3\}$, the linear relaxation of $x_i + x_j \leq 1$ for all $1 \leq i < j \leq 3$ can also be satisfied by, e.g., $x_1^* = x_2^* = x_3^* = 0.5$; this solution x^* would be cut off by the *clique cut* $x_1 + x_2 + x_3 \leq 1$.

In general, given a relaxed solution $x^* \in [0, 1]^n$, we want to find a set C of mutually exclusive indices that maximizes $\sum_{i \in C} x_i$; we have found a violated clique constraint if that sum is (sufficiently) larger than 1. This is an instance of the (weighted) clique problem if we consider the items to be vertices of a graph connected by an edge if two items are mutually exclusive.

Implement a heuristic that finds such violated constraints.[2] For instance, you can use a greedy heuristic based on sorting the items by non-increasing x_i^* and building cliques greedily in the resulting order, starting from some (e.g., the k largest x_i^* for some k) number of starting points.

Apart from these issues, there are further problems that plague this implementation, such as a pure DFS tree exploration and naïve branch variable selection; we will address these issues in a later exercise.

Exercise 2 (Exam Scheduling): **(15 points)**

A faculty has to schedule the written exams for the end of a term. The planning horizon is a fixed list of weekdays — Saturdays and Sundays are not used. Each weekday offers a small, heterogeneous set of rooms with fixed seat counts: typically one large lecture hall, a couple of mid-sized rooms and one or two small rooms. Every student is enrolled in a fixed list of exams that they have to write. The faculty has to place each exam into exactly one room on exactly one of the available days; the chosen room has to seat all of the exam's enrolled students. A given room on a given day can host more than one exam — different sittings, e.g. a morning and afternoon block — but only up to a per-row limit included in the rooms table (typically between one and four sittings). No student may write two exams on the same day, and no student may write more than three exams within any single ISO calendar week. Within those limits, the faculty would like the exams of any individual student to spread out across weeks rather than pile up: every week in which a student writes more than one exam contributes one penalty point per extra exam, and the goal is to minimize the total penalty summed over all students and weeks.

The companion webapp at `https://ae.krupke-algorithms.de/` hosts the benchmark

[2] The method stub is called `clique_cut_separation`.

instance, the Pydantic schema for the two raw tables — a slots table with rows `room` | `date` | `capacity` | `max_exams` and an enrollments table with rows `student` | `exam` —, and a visualizer that renders any solution you paste in. The dates are real `datetime.date` values; weeks are derived as ISO calendar weeks.

Model the problem with Google OR-Tools' CP-SAT solver. Hand in the source code you wrote together with a single solution file `my_solution.json` produced by your best CP-SAT run, in the `Solution` format from the schema. We will run your code, validate the solution against the rules, and read its objective value.

Exercise 3 (Bike Redistribution): **(15 points)**

A bike-sharing operator has to rebalance its fleet over the course of a single shift. One vehicle starts the shift at the central depot, visits a list of bike stations across the city, and returns to the depot when the run is over. Some of those stations have accumulated more bikes than they should and need a pickup; the rest have run dry and need a drop-off. Pickup and drop-off stations come in equal numbers, so a complete shift moves every surplus bike to a station that needs one. The vehicle is small: it can carry at most a handful of bikes at any moment, which means the driver has to interleave pickups and drop-offs along the route rather than collect everything first. Any pickup contributes a bike to the load, any drop-off takes one off, and the vehicle is never allowed to be over capacity nor to attempt a drop-off while empty. Because every pickup station has exactly one bike to hand off and every drop-off station can absorb exactly one bike, it does not matter which surplus bike ends up at which deficit station — only that the totals match by the end. The vehicle returns to the depot with an empty rack. The operator wants the shift to be as short as possible, measured by the total distance the vehicle travels along its route.

The companion webapp at `https://ae.krupke-algorithms.de/` hosts the benchmark instances, the Pydantic schema for the two raw tables — a locations table with rows `name` | `kind` | `lat` | `lon` and a travel-times table with rows `src` | `dst` | `distance` — along with the vehicle capacity, and a visualizer that renders any solution you paste in. The travel-times table is asymmetric in general.

Model the problem with Google OR-Tools' CP-SAT solver. Hand in the source code you wrote together with one solution file per benchmark instance. Each file is named `my_solution_<NN>.json`, where `<NN>` is the two-digit instance number (`01` through `10`) from the webapp, in the `Solution` format from the schema. We will run your code, validate every solution against the rules, and read its tour distance.