Algorithm Engineering **Summer 2026**

Dr. Phillip Keldenich
Dr. Dominik Krupke

Submission due: 18.06.2026 16:45

Homework 4

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 (Crate Booking): **(12+8+20+10 points)**

A logistics company runs trucks between D depots over a horizon of T days. *Empty* crates accumulate where deliveries end and are needed where the next deliveries start, so on each day a depot has a net *surplus* or a net *shortage* of empty crates. The crates already exist; they are simply in the wrong place at the wrong time and must be redistributed.

We model this on a **time-expanded graph**: one node per pair (depot, day), written $v = (p, t)$. Let $b_v \in \mathbb{Z}$ be the net balance at node v ($b_v > 0$ surplus, $b_v < 0$ shortage), with $\sum_v b_v = 0$. Crates move along directed arcs from day t to day $t + 1$:

- **free arcs** A_{free}, each with capacity $c_a \geq 0$ and *zero* cost: an overnight *hold* at a depot ($p \to p$), or a *piggyback* arc carrying crates on the spare slots of a truck that already drives that lane that day. Which lanes have spare slots, and how many, changes from day to day.
- **bookable trucks** A_{book}: for a lane/day you may *book* extra dedicated trucks. Booking $y_a \in \mathbb{Z}_{\geq 0}$ trucks on arc a adds $y_a \cdot \kappa_a$ crate slots and costs $y_a \cdot \varphi_a$, where κ_a is the per-truck capacity and φ_a is the fixed booking fee (proportional to the haul distance).

Demand is **soft**: covering every shortage is not mandatory. Each crate of shortage left unserved costs a fixed **unmet penalty** π, so the planner may always pay π per crate instead of booking. This is what frees the instances from needing to be fully coverable, and turns Part 2 into a genuine trade-off rather than a forced cover.

The companion webapp at `https://ae.krupke-algorithms.de/` hosts the benchmark instances, the Pydantic schema for the raw tables — a depots table with rows `name` | `lat` | `lon`, a balances table `depot` | `day` | `net`, a free-arcs table `src` | `src_day` | `dst` | `dst_day` | `capacity` | `kind` (`hold` or `piggyback`), and a truck-options table `src` | `dst` | `day` | `capacity` | `fixed_cost`, together with the per-crate unmet penalty π — and a visualizer that renders any solution you paste in. The instance is the data a planner would hand over; the modeling is yours.

Tooling. You need only a handful of library calls; everything else is your own modeling. **Part 1** is a `networkx` max flow plus the matching min cut:

```
import networkx as nx
g = nx.DiGraph()
g.add_edge("s", ("Kassel", 1), capacity=7)   # nodes may be any hashable, e.g. a (
    depot, day) tuple
g.add_edge(("Kassel", 1), "t", capacity=5)
flow_value, flow_dict = nx.maximum_flow(g, "s", "t")   # flow_dict[u][v] = units on
    arc (u, v)
cut_value, (reachable, unreachable) = nx.minimum_cut(g, "s", "t")  # cut_value ==
    flow_value
# an arc (u, v) is saturated on the cut iff u in reachable and v in unreachable
```

Part 2 is a MIP built with OR-Tools' MathOpt and solved by HiGHS:

```
from datetime import timedelta
from ortools.math_opt.python import mathopt
model = mathopt.Model(name="crate_booking")
f = model.add_integer_variable(lb=0, ub=cap, name="f")  # integer var; add_variable
    () is continuous
y = model.add_integer_variable(lb=0, name="y")
model.add_linear_constraint(f <= capacity * y)          # big-M activation link
model.add_linear_constraint(sum(out_arcs) - sum(in_arcs) == net)  # flow
    conservation
model.minimize(sum(fee * y_t for ...) + penalty * sum(u_n for ...))
res = mathopt.solve(model, mathopt.SolverType.HIGHS,
                    params=mathopt.SolveParameters(time_limit=timedelta(seconds=30))
    )
res.termination.reason == mathopt.TerminationReason.OPTIMAL   # proven optimal?
vals = res.variable_values(); units = round(vals[f])          # read a variable's
    value
```

For the LP relaxation in the last part, build the booking variables with `add_variable` instead of `add_integer_variable`; nothing else changes.

a) **Part 1 — Max flow (for free).** Ignore booking. Model "how many crates can reach a shortage using only the free arcs" as a max flow: add a super-source s with arcs $s \to v$ of capacity b_v for every surplus node, a super-sink τ with arcs $v \to \tau$ of capacity $-b_v$ for every shortage node, and every free arc with its capacity c_a. Solve it with a max-flow library (e.g. `networkx`) and report the maximum flow, i.e. the number of crates that can be redistributed at no cost.

b) **The bottleneck.** Compute a **minimum cut** and report the gap $\big(\sum_v \max(0, -b_v)\big) -$ maxflow, the crates that *cannot* be delivered for free. Which free arcs are saturated on the cut, and why does the min cut tell the planner where an extra truck would help most? (One or two sentences.)

c) **Part 2 — Fixed-charge MIP (book or pay).** Now allow booking, but keep demand soft: cover shortages where it pays, and pay the penalty π per crate where it does not. Model the problem as a MIP (e.g. with Google OR-Tools' MathOpt and the HiGHS backend) and minimize booking fees plus the unmet penalty. The twist is the **fixed charge**: a booked truck's fee φ_a is paid in full the moment it carries a single crate, so a plain capacitated flow cannot express it. *Hint:* you will

need a **big-M** style constraint that links the integer number of trucks booked on a lane to the flow that lane may carry; the per-truck capacity κ_a is the natural big-M here. Hand in the resulting plan for every benchmark instance as a solution file (see below); the objective, the trucks you book, and the demand you leave unmet are all read from it.

d) **Why a MIP?** Solve the *LP relaxation* (drop the integrality of the booking variables). Report its objective and explain why it is a strict lower bound here and what its fractional booking values mean. Briefly: which single feature turns the polynomial Part 1 into the NP-hard Part 2?

Report your results as the actual solution files: one `my_solution_<NN>.json` per benchmark instance, where `<NN>` is the two-digit instance number from the webapp, in the `Solution` format from the schema. Hand them in together with the source code you wrote and short written answers to the bottleneck and "why a MIP" questions above. We validate every solution against the rules and read each instance's objective directly from your submitted file.