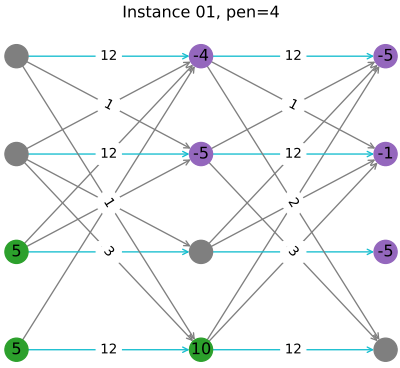
Technische
Universität
Braunschweig

Algorithm Engineering 2026: Sheet 04 & 05

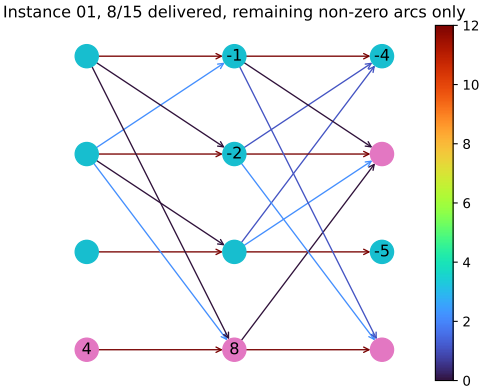
Rouven Kniep, July 02, 2026

HW04 (Crate Booking) Pt1: Setting



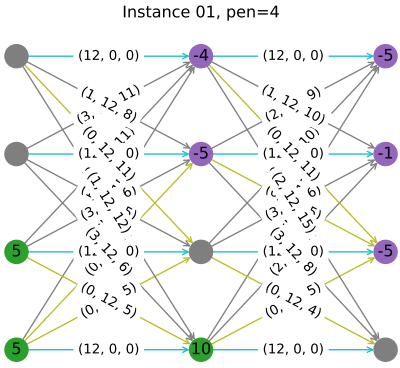
- **Depots** (rows)
- **Days** (columns)
- **Node**: days $\times$ depots
- **Balance**: surplus / demand at some nodes
- **free Arcs**: node to node with capacity (1st element)
- **Trucks**: not available! free flow only
- **Unmet Penalty**: penalty for each crate of unmet demand

HW04 (Crate Booking) Pt1: Solution



- omitted SuperSource & SuperSink
- **Max Flow**: (one) allocation (edge→flow) with maximum total flow
- **Min Cut**: (one) 2-partition of nodes (before the cut, after the cut)
- all forward edges across cut: fully utilized $\implies$ more cap/trucks **might** help
- edges within a partition (or backwards across cut): more capacity **cannot** help
- **Planners**: book trucks across cut towards underutilized edges, insert + reevaluate

HW04 (Crate Booking) Pt2: Setting



- **Depots** (rows)
- **Days** (columns)
- **Node**: days $\times$ depots
- **Balance**: surplus / demand at some nodes
- **free Arcs**: node to node with capacity (1st element)
- **Trucks**: node to node, capacity (2nd) & cost (3rd) per truck
- **Unmet Penalty**: penalty for each crate of unmet demand

Crate Booking Model

Idea: use graph as data structure

- can store any data in nodes / edges
- available query-operations
- native to our problem
- partially implemented from Pt.1

problem native:

- for each edge...
  - flow: decision var
  - booked trucks: decision var
  - capacity: constraint
  - obj+= truck costs
- for each node...
  - flow conservation: constraint
  - obj+= unmet demand penalty
- minimize objective penalty

Edge Modeling

```
1  def _model_edge_flow(self):     # model edge flow + trucks
2    for src, dst, data in self.g.edges(data=True):
3      data["flow_var"] = self.var(lb=0, name=f"flow_{src}_{dst}")
4      if data.get("truck_cap", 0) != 0:
5        data["truck_var"] = self.intvar(lb=0, name=f"trucks_{src}_{dst}")
6        self.model.add_linear_constraint(
7          data["flow_var"] <= data.get("capacity", 0)
8          + data["truck_cap"] * data["truck_var"])
9        self.objective += data["truck_var"] * data.get["truck_cost"]
10     else:
11       self.model.add_linear_constraint(
12         data["flow_var"] <= data.get("capacity", 0))
```

Node Modeling[1]

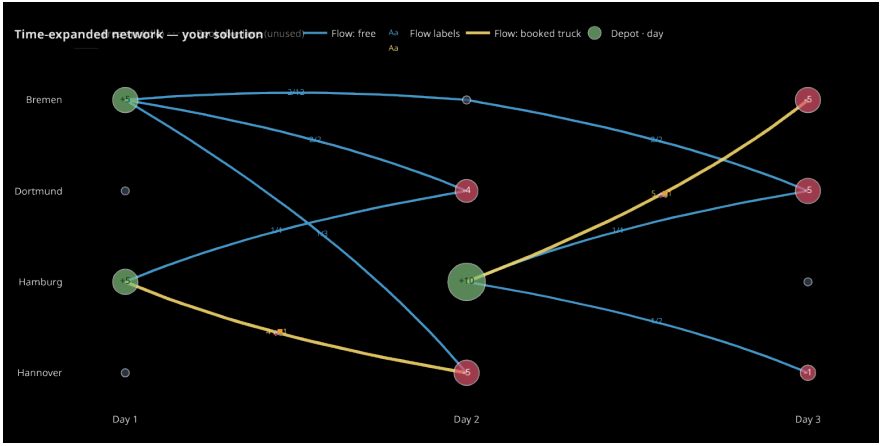
```
1  def _model_node_flow(self):
2    for node, ndata in self.g.nodes(data=True):
3      incoming = [var for src, dst, var
4        in self.g.in_edges(nbunch=node, data="flow_var")]
5      outgoing = [var for src, dst, var
6        in self.g.out_edges(nbunch=node, data="flow_var")]
7      delta = ndata.get("net", 0)
8      flow = sum(incoming) - sum(outgoing)
```

"problem native" data structure: query operations!

Node Modeling[2]

```
1     # node flow equality: in - out + unmet_pen + delta >= 0
2     if delta > 0:
3       c = flow + delta >= 0 c # allow not picking up
4     elif delta < 0:
5       ump = self.var(lb=0, ub=-delta, name=f"unmet_pen_{node}")
6       self.objective += ump * self.instance.unmet_penalty
7       c = flow + ump + delta == 0
8     else:
9       c = flow == 0
10    self.model.add_linear_constraint(c, name=f"node_flow_{node}")
```

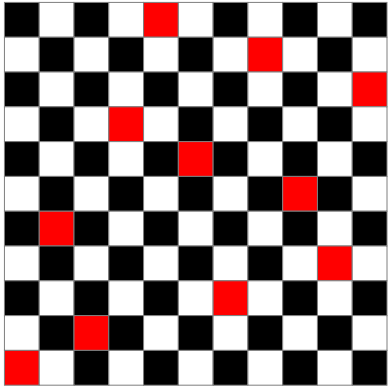
HW04 (Crate Booking) Pt2: Example Solution



HW04 (Crate Booking) Pt2: Linear Programming Relaxation

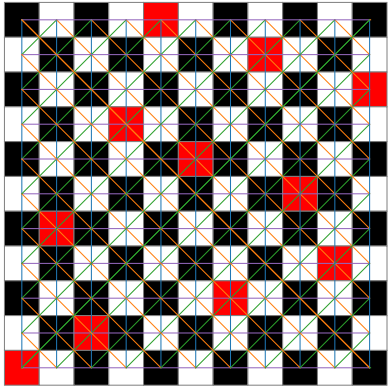
- "solve LP relaxation": replace **all** integer variables with continuous
- "explain why it is a *strict* lower bound"
  - $\texttt{relaxation} \subseteq \texttt{integer model} \wedge \texttt{minimization} \implies \texttt{lower bound}$
  - "*strict*": doesn't say anything: objective can be the same!
- "Why a MIP?"
  - partial usage of a truck $\implies$ full payment: $\texttt{flow} \leqslant \texttt{capacity} + \texttt{truck\_cap} * \texttt{truck\_var}$
  - therefore `truck_var` must be non-continuous integer - rest can be continuous and linear
  - LP (P-time) $\rightarrow$ MIP (NP-hard)

HW05 Ex1 Setting: n Queens



- $n * n$ chess grid
- n queens
- position queens on the chess field
- no queen can see another

n Queens: Constraints (visualized)



Constraints:

- exactly one per column
- exactly one per row
- at most one per "positive" diagonal
- at most one per "negative" diagonal

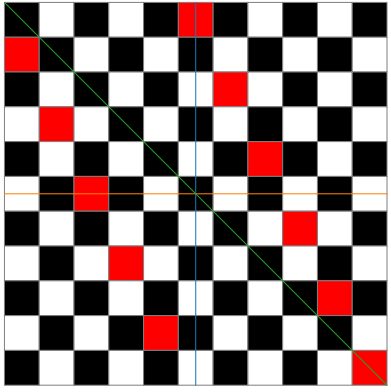
n Queens: Constraints (formalized)

columns (rows similar)	cardinality	$\sum_{y=0}^{n-1} b_{xy} = 1$	$\forall x \in [0, n-1]$
	SAT (CNF)	$\bigvee_{y=0}^{n-1} b_{xy}$	$\forall x \in [0, n-1]$
		$\neg b_{xy} \vee \neg b_{iy}$	$\forall i, x, y \in [0, n-1]^3 \mid i \neq x$
positive diagonal (negative similar)	cardinality	$(\sum b_{xy} \mid x - y = k) \leqslant 1$	$\forall k \in [-n+2; n-2]$
	SAT (CNF)	$\neg b_{ij} \vee \neg b_{xy}$	$\forall x, y, i, j \in [0, n-1]^4$ $\mid i + j = x + y \wedge i \neq x$

n Queens: Constraints implementation

```
1  self.cnf.extend(list(self.iter_row_lits(row)) for row in range(self.n))
2  self.cnf.extend(list(self.iter_col_lits(col)) for col in range(self.n))
3  for row in range(self.n):
4    self.cnf.extend((-l1, -l2) for l1, l2 in self.iter_row_combinations(row))
5  for col in range(self.n):
6    self.cnf.extend((-l1, -l2) for l1, l2 in self.iter_col_combinations(col))
7  for k in range(2 * self.n):  # "sum" diagonal
8    self.cnf.extend((-l1, -l2)
9      for l1, l2 in self.iter_neg_diag_combinations(k))
10 for k in range(-self.n + 1, self.n):  # "difference" diagonal
11   self.cnf.extend((-l1, -l2)
12     for l1, l2 in self.iter_pos_diag_combinations(k))
```

n Queens: Symmetry Breaking



Symmetries:

- horizontal: row 0 queen must be in right half
- vertical: row 0 queen must be right of row n-1 queen
- diagonal: row 0 queen column number must be equal or lower than column 0 queen row number
- ... more possible (positive diagonal)

n Queens: Symmetry Breaking implementation

```
1  self.cnf.append([self.get_lit(0, col) for col in range(ceil(self.n / 2))])
2
3  for topcol in range(1, ceil(self.n / 2)):
4    toplit = self.get_lit(0, topcol)
5    # vertical
6    self.cnf.extend([-toplit, -self.get_lit(self.n - 1, bottomcol)]
7      for bottomcol in range(topcol))
8
9    # negative diagonal
10   for col in range(1, ceil(self.n / 2)):
11     toplit = self.get_lit(0, col)
12     self.cnf.extend([-toplit, -self.get_lit(row, 0)]
13       for row in range(col))
```

n Queens: CP-SAT cardinality

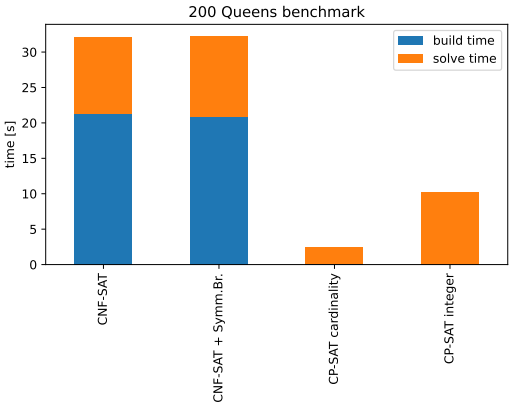
```python
# rows/cols: exactly one
for row in range(self.n):
  self.model.add_exactly_one(self.iter_row_lits(row))
for col in range(self.n):
  self.model.add_exactly_one(self.iter_col_lits(col))

# diagonals: at most one
for row in range(0, self.n):
  self.model.add_at_most_one(self.iter_diagonal_lits(row, 0, positive=True))
  self.model.add_at_most_one(self.iter_diagonal_lits(row, 0, positive=False))
for col in range(0, self.n):
  self.model.add_at_most_one(self.iter_diagonal_lits(0, col, positive=True))
  self.model.add_at_most_one(self.iter_diagonal_lits(self.n - 1, col, positi
```

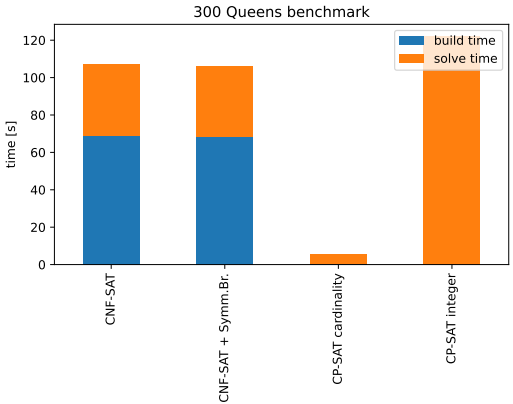
n Queens: CP-SAT integers (official version!)

```python
# one per row -> var
self.vars = [self.model.new_int_var(0, self.n - 1, name=f"q{row}")
  for row in range(self.n)

self.model.add_all_different(self.vars)  # one per column

# positive diagonals
self.model.add_all_different([var + row
  for row, var in enumerate(self.vars)])

# negative diagonals
self.model.add_all_different([var - row
  for row, var in enumerate(self.vars)])
```

n Queens: Benchmarks

Limits: CNF-SAT 350 (memory, 160s), CP-SAT int 350 (time), CP-SAT card 1500 (memory, 60s)





HW05 Ex2a: CNF Bounded Variable Elimination Correctness

" $\implies$ " (φ SAT $\implies$ φ' SAT):

- Clauses in φ' that are in φ are naturally satisfied.
- All other clauses in φ' are of form $A \vee B$ as a result of resolving $(x \vee A) \wedge (\neg x \vee B) \in \varphi$.
- If $x \in S$: as $S \models \neg x \vee B$, then $S \models B$, which implies $S \models A \vee B$ and therefore $S \models \varphi'$
- If $\neg x \in S$: as $S \models x \vee A$, then $S \models A$, which implies $S \models A \vee B$ and therefore $S \models \varphi'$

" $\impliedby$ " (S' SAT $\implies$ S' SAT):

- let $S_+ = S' \cup x$ and $S_- = S' \cup \neg x$
- W.l.o.g. let $S_+ \not\models \varphi$: reason must be $\neg x \vee B = \bot$ that was removed and is not satisfied by $x = \top$
- $S' \models A \vee B$ forall A as $S' \not\models B$, therefore $S_- \models \varphi$:
- as $S' \models A \implies S' \models x \vee A$, and $S_- \models \neg x \vee B$ naturally.

HW05 Ex2b: CNF Vivification

let $C = l_1 \vee ... \vee l_k$, with propagation $P = \neg l_1 \wedge ... \wedge \neg l_j$

1. $P \implies l_i \mid k \geqslant i \geqslant j$: C can be removed, as $(P \wedge l_i) \models C$.
   in practice: not necessarily effective!
2. $P \implies \neg l_i \mid k \geqslant i \geqslant j$: C can be strengthened into C' by removing l_i.
   That would change satisfiability of S, iff $c_i = \top \wedge \forall 1 \leqslant l \leqslant k \wedge l \neq i : c_l = \bot$, but $\neg l_1 \wedge ... \wedge \neg_k \implies \neg c_i$, which contradicts.
3. $\neg l_1 \wedge ... \wedge \neg l_j$ causes a conflict: we learn $N = l_1 \vee ... \vee l_j \subset C$ and can remove C.

HW05 Ex2c: CNF Blocked Clause Elimination

let $C = l \vee a_1 \vee ... \vee a_k \in \varphi$.
C is *blocked on* l if $\forall D \mid \neg l \in D : C \diamond_l D \implies \top$ and can then be removed
" $\impliedby$ " (φ SAT $\implies$ φ' SAT): trivial, as $\forall P \mid P \models \varphi \implies P \models \varphi'$.
" $\implies$ " (φ' SAT $\implies$ φ SAT): let $S' \models \varphi'$

- If $S' \models \varphi$: done.
- If $S' \not\models \varphi$, then $S' \not\models C$ and therefore $\neg l \in S'$.
- let $S = (S' \setminus \neg l \cup l)$, then $S \models C$, but might not model a D.
- however $\forall D : \exists a_i \mid 1 \leqslant i \leqslant k : \neg a_i \in D$, which is given by above.
- therefore $\forall D : S \models D$