Technische
Universität
Braunschweig

Algorithm Engineering 2026: Sheet 03

Rouven Kniep, June 04, 2026

evaluation

https://umfragen.tu-bs.de/evasys/online.php?pswd=9728N

Ex1: Setting

Mutually Exclusive Knapsack

Given: Instance with¨

- Items $X = \{x_1, \ldots, x_n\}$
- Profits $p_i \in \mathbb{R}_{\geqslant 0}$
- Weights $w_i \in \mathbb{R}_{\geqslant 0}$
- Capacity $c \in \mathbb{R}_{\geqslant 0}$
- Excluded items $E_i \subseteq X$ (symmetrical)

Constraints and Target

- find solution $S \subseteq X$ with:
- $max \sum_{i=1}^{n} p_i * x_i$ (optimization function), subject to:
- $\sum_{i=1}^{n} w_i x_i \leqslant c$ (within capacity)
- $\forall i : \forall j \in E_i : x_i + x_j \leqslant 1$ (mutually exclusive)
- $\forall i : x_i \in \mathcal{B} = \{0; 1\}$ (binary variables)

Ex1a: Constraint Propagation (overview)

Terminology

what?	best found solution	best possible value
validity	global	local (node)
min.	upper bound	lower bound
max.	lower bound	upper bound
Gurobi	incumbent	bound
CP-SAT	objective	bound

In each node:

1. dequeue node
2. if incumbent $\geqslant$ bound: next node
3. **constraint propagation** - extend node's partial solution
4. get bound + improve bound with cuts
5. update stats
6. if incumbent $\geqslant$ bound: next node
7. otherwise: generate + enqueue branches

Ex1a: Constraint Propagation

given: partial assignments P: `list[bool|None]`

---

constraints: mutually exclusive - w.l.o.g. let $x_i + x_j \leqslant 1$ and $P_i \in$ `bool`

- if P_i=`False`: constraint unnecessary
- if P_i=`True`: boolean case: $\implies$ P_j = `False`; fractional case: $\implies$ $x_j^* = 0$
- $\implies$ no gain to be had

---

constraint: capacity $\sum_{i=1}^{n} w_i x_i \leqslant c$

- let $rem = c - \sum_{i=1}^{n} (w_i \mid P_i = \texttt{False})$
- w.l.o.g let $\exists j \mid w_j < rem$
- boolean case: $\implies$ P_j=`False`
- fractional case: $\implies$ $x_j^* \in [0; 1)$
- $\implies$ relaxation may assume impossible values $\implies$ propagate!

Ex1a: Constraint Propagation (solution)

```python
def constraint_propagation(self, partial_assignment: list[bool | None]):
  remaining_capacity = self.instance.capacity - sum(
    self.index_to_item[i].weight
    for i, assigned in enumerate(partial_assignment)
    if assigned
  )
  for i, assigned in enumerate(partial_assignment):
    if assigned is None:
      if self.index_to_item[i].weight > remaining_capacity:
        partial_assignment[i] = False
```

Ex1a: Constraint Propagation (wrong solution)

```python
def constraint_propagation(self, partial_assignment: list[bool | None]):
  remaining_capacity = self.instance.capacity
  for i, assigned in enumerate(partial_assignment)
    if assigned is None:
      if self.index_to_item[i].weight >= remaining_capacity:
        partial_assignment[i] = False
    elif assigned is True:
      remaining_capacity -= self.index_to_item[i].weight
```

Ex1a: Constraint Propagation (wrong solution)

```
1  def constraint_propagation(self, partial_assignment: list[bool | None]):
2    remaining_capacity = self.instance.capacity
3    for i, assigned in enumerate(partial_assignment)
4      if assigned is None:
5        if self.index_to_item[i].weight >= remaining_capacity:
6          partial_assignment[i] = False
7      elif assigned is True:
8        remaining_capacity -= self.index_to_item[i].weight
```

- wrong comparison (ln. 5): would disable fitting items
  $\Longrightarrow$ INCORRECT! Algorithm will throw out valid solutions!
- merged loops: `remaining_capacity` is decreased after propagation check
  $\Longrightarrow$ valid, but too weak: does not disable all eligible items

Ex1b: Heuristic

```
1  for rd in range(ROUNDS): # ROUNDS = 1000 - still 0.1s!
2    result = [False] * len(self.instance.items)
3    candidates = sorted_idxs.copy(); excludes = set()
4    profit = 0.0; remaining_cap = self.instance.capacity
5    while candidates:
6      num_choices = min(len(candidates), NUM_CANDIDATES)
7      i = random.choice(candidates[:num_choices])
8      candidates.remove(i); item = self.item_to_index[i]
9      if item.weight <= remaining_cap and item.name not in excludes:
10       remaining_cap -= item.weight; profit += item.profit
11       result[i] = True; excludes.update(item.excludes)
12   if profit > best_profit:
13     best_profit = profit; best_result = result
```

Ex1c: Knapsack Cuts

Flow:

1. calculate relaxation
2. if relaxation is int: return
3. if `cut_rounds++` > `max_cut_rounds`: return
4. try to find cuts
5. if no cuts found: return
6. add cuts + repeat

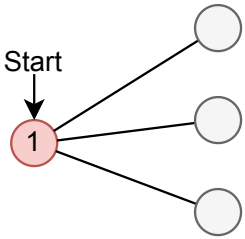
Idea: find $D \in [1; n]$, s.t.

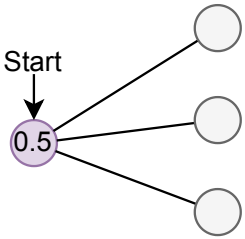
1. $\sum_{i \in D} w_i > c$
2. $\sum x_i^* > |D| - 1$

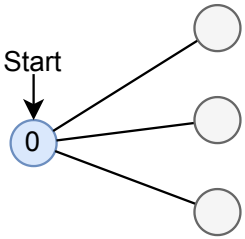
Ex1c: Knapsack Cuts

```
best_cut_indices = None; best_violation = 0.0
for sorted_indices in (sorted_indices_1, sorted_indices_2):
  weight_sum = 0.0; value_sum = 0.0; cut_indices = []
  for i in sorted_indices:
    weight_sum += self.index_to_item[i].weight
    value_sum += solution[i]; cut_indices.append(i)
    violation = value_sum - (len(cut_indices) - 1) - CUT_VIOLATION_TOLERANCE
    if weight_sum > capacity_with_buffer:
      if violation > best_violation:
        best_violation = violation
        best_cut_indices = cut_indices.copy()
      break
    if violation <= best_violation:
      break
```

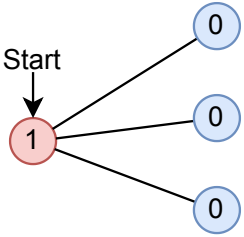
Ex1d: Clique Cuts - which items to include



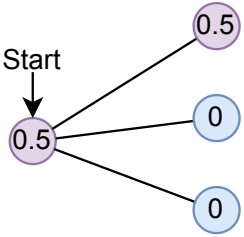




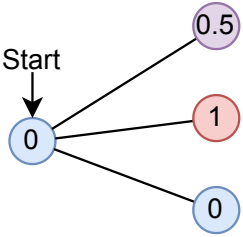
Ex1d: Clique Cuts - which items to include



all candidates have value 0
$\Longrightarrow$ no cuts

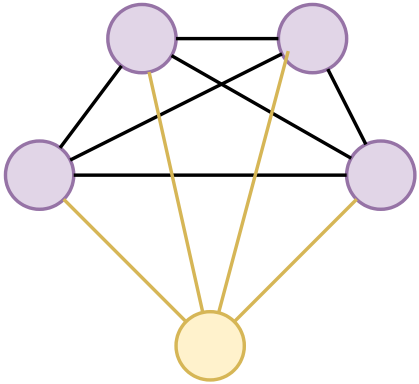


candidates have values
0-0.5 $\Longrightarrow$ interesting!



unlikely (interesting cliques found from 2.)

Ex1d: Clique Cuts - Clique Size



- avoid items of value 1
- maximum clique constraints imply all smaller ones:

$$x_1^* + x_2^* + x_3^* + x_4^* + x_5^* \leqslant 1$$
$$\implies x_1^* + x_2^* + x_3^* + x_4^* \leqslant 1$$

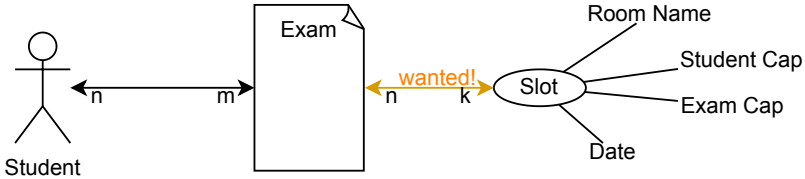
Ex1d: Clique Cuts

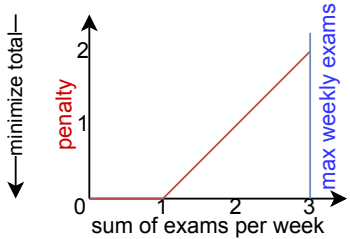
```python
def clique_cut_separation(self, solution: Sequence[float]) -> list[int] | No
  sol = list(solution)
  candidates = [i for i, value in enumerate(sol)
    if (value > INTEGRALITY_TOLERANCE
    and value < 1 - INTEGRALITY_TOLERANCE)]
  candidates.sort(key=lambda i: sol[i], reverse=True)
  best_clique: list[int] = []; best_value = 0.0
  # inner loop, see next slide
  if best_value <= 1.0 + CUT_VIOLATION_TOLERANCE:
    return None
  else:
    return best_clique
```

Ex1d: Clique Cuts

```
1    # candidates are without 1 and 0 -> all are interesting
2    for anchor in candidates[:20]:
3      clique = [anchor]; value = solution[anchor]
4      extensions = self.exclusion_graph[anchor].copy()
5      while extensions:
6        best = min(extensions, key=lambda i: solution[i])
7        clique.append(best); value += solution[best]
8        extensions.intersection_update(self.exclusion_graph[best])
9      if value > best_value:
10       best_value = value; best_clique = clique.copy()
```

Ex2: Exam Scheduling - Setting





max 1 exam per day foreach student

limit of exams per slot

all exams scheduled

all exams scheduled

exam students must fit slot student cap

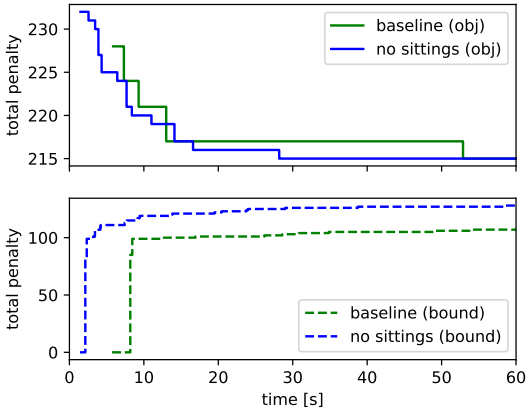
Variables per Sitting? (wrong!)

```
1  def _make_vars(self):
2   self.vars = {
3     (exam, slot, sitting):
4     self.model.new_bool_var(
5       name=f"({exam},{slot},{sitting})"
6     )
7     for exam in self.exams
8     for slot in self.instance.slots
9     for sitting in range(slot.max_exams)
10   }
```

- Idea: assign each sub-slot in exams per day individually
- so: morning/noon/afternoon/evening...
- but: not necessary (we don't need the exact sub-slot for anything)

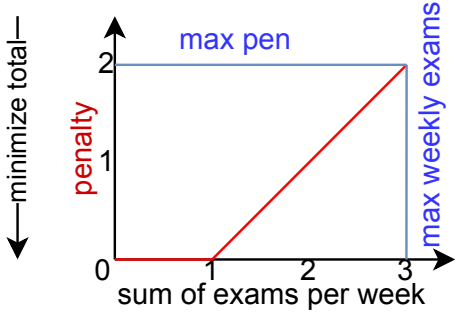
Exam-Slot-Variables (good!)

```
1  def _make_vars(self):
2  self.vars = {
3    (exam, slot):
4    self.model.new_bool_var(
5    name=f"({exam},{slot})"
6    )
7    for exam in self.exams
8    for slot in self.instance.slots
9  }
```



$\Longrightarrow$ only model needed complexity

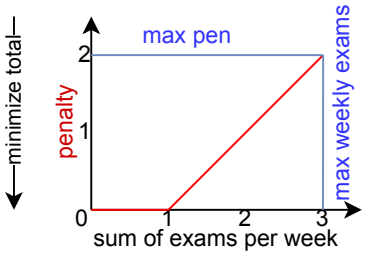
Objective

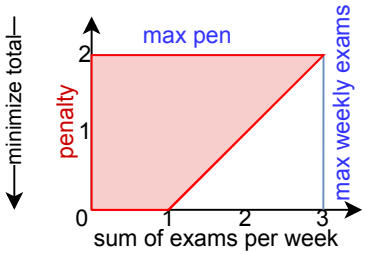


- penalty 0 if sum of exams $\leqslant 1$, $+1$ for each more
- not linear! $\Longrightarrow$ additional `penalty`-variables needed

```
for student, exams in self.students.it
  for isoweek, slots in self.isoweeks
    penvar = self.model.new_int_var(
      0, 3 - 1,
      f"pen_{student}_{isoweek}")
    self.model.add_max_equality(
      penvar,  0, sum(
        self.vars[(exam, slot)]
        for exam in exams
        for slot in slots
      )- 1)
      self.penalties.append(penvar)
self.model.minimize(sum(self.penalties
```

Objective (improved)

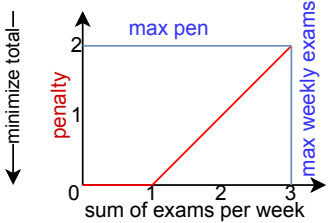


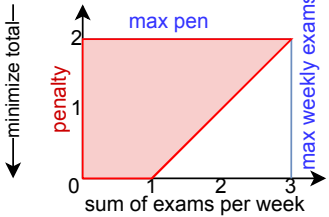


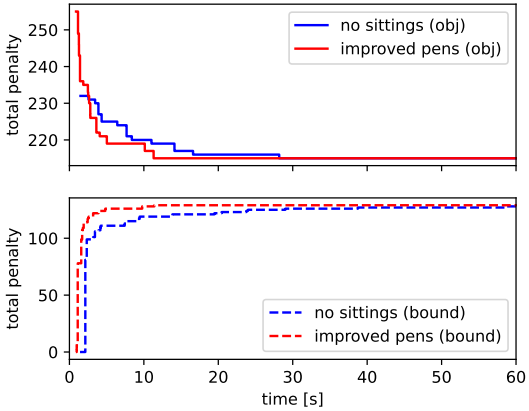

```python
self.model.add_max_equality(
  penvar,  0, sum(
  self.vars[(exam, slot)]
  for exam in exams
  for slot in slots
) - 1)
```

```python
self.model.add(penvar + 1
  >= sum(self.vars[(exam, slot)]
    for exam in exams
    for slot in slots)
)
```

Objective (improved)







$\Longrightarrow$ avoid unnecessary non-convex constraints

Room constraints

```python
def _exam_constraints(self):
  # Capacity
  for (exam, slot), var in self.vars.items():
    self.model.add(var * self.exams[exam] <= slot.capacity)
  # Held once
  for exam in self.exams:
    self.model.add_exactly_one(
      self.vars[(exam, slot)] for slot in self.instance.slots
    )
```

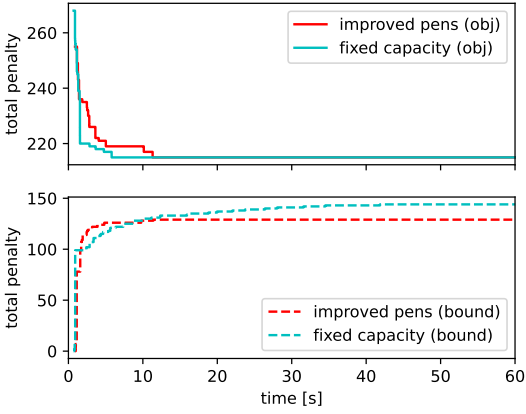
Room constraints

```
1  def _exam_constraints(self):
2    # Capacity
3    for (exam, slot), var in self.vars.items():
4      self.model.add(var * self.exams[exam] <= slot.capacity)
5    # Held once
6    for exam in self.exams:
7      self.model.add_exactly_one(
8        self.vars[(exam, slot)] for slot in self.instance.slots
9      )
```

- relaxes poorly: fractional values caused by room cap
  (example: 45 cap and 50 students $\implies$ var$\leqslant 0.9$)
- room cap and number of students known during model creation
  $\implies$ could force variable to 0 or not even make variable

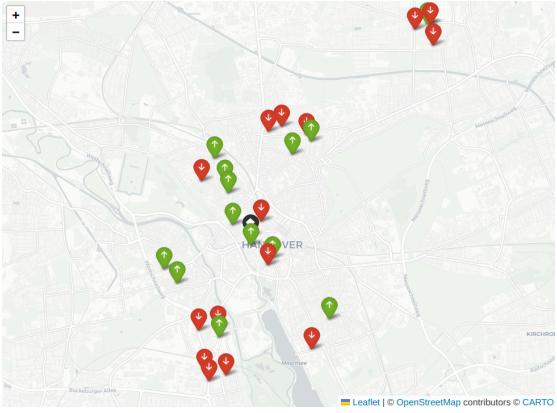
Room Constraints (improved)

```
1  def _make_vars(self):
2    self.vars = {(exam, slot):
3      self.model.new_bool_var(
4        name=f"({exam},{slot})")
5      for exam, num_students
6        in self.exams.items()
7      for slot in self.instance.slots
8      if num_students <= slot.capacity
9    }
10   # and a check that all exams have
11   # at least one feasible slot
```



$\Longrightarrow$ only model constraints and variables about data not known during construction!

Exercise 3: Bike Redistribution



- bikes: pick up and drop off at locations
- vehicle with max capacity
- tour from and to depot
- distances: given matrix
- wanted: shortest tour

Decision Variables

```python
def _make_vars(self):
  # arcs: directed edge from node *u* to *v*
  # directed for distance, also performs action at *v*
  self.arc_vars = {
    (u, v): self.model.new_bool_var(f"arc[{u}->{v}]")
    for (u, v) in itertools.permutations(self.all_nodes, 2)
  }
  # load: loaded bikes after stop at node *n*
  self.load_vars = {
    n: self.model.new_int_var(0, self.instance.capacity, f"load[{n}]")
    for n in self.all_nodes
  }
```

Objective

```python
def _objective(self):
  # min total travel distance
  self.model.minimize(
    sum(self.distances[u, v] * lit
    for (u, v), lit in self.arc_vars.items())
  )
```

- perfectly linear
- no additional helper-variables necessary
- good relaxations...

Capacity Constraints

```python
def _capacity_link_constraints(self):
    # Channelling: along any used arc (u, v): load[v] = load[u] + delta[v].
    self.model.add(self.load_vars[self.depot] == 0)
    for (u, v), lit in self.arc_vars.items():
        if v == self.depot:
            self.model.add(self.load_vars[u] == 0).only_enforce_if(lit)
        else:
            self.model.add(
                self.load_vars[v] == self.load_vars[u] + self.delta[v]
            ).only_enforce_if(lit)
```

- directly based on connection $\implies$ no helpers / no complex implications

Circuit Constraints (MTZ - Bad)

```
1  def _circular_constraint_mtz(self):
2    self.order_vars = {n: self.model.new_int_var(
3      lb=0, ub=len(self.all_nodes) - 1, name=f"order_{n}")
4      for n in self.all_nodes}
5    self.model.add(self.order_vars[self.depot] == 0)
6    self.model.add_all_different(self.order_vars.values())
7    for (u, v), lit in self.arc_vars.items():
8      target = len(self.all_nodes) - 1 if v == self.depop
9          else self.order_vars[v] - 1
10     self.model.add(self.order_vars[u] == target).only_enforce_if(lit)
11   # + one incoming + one outgoing for every node
```

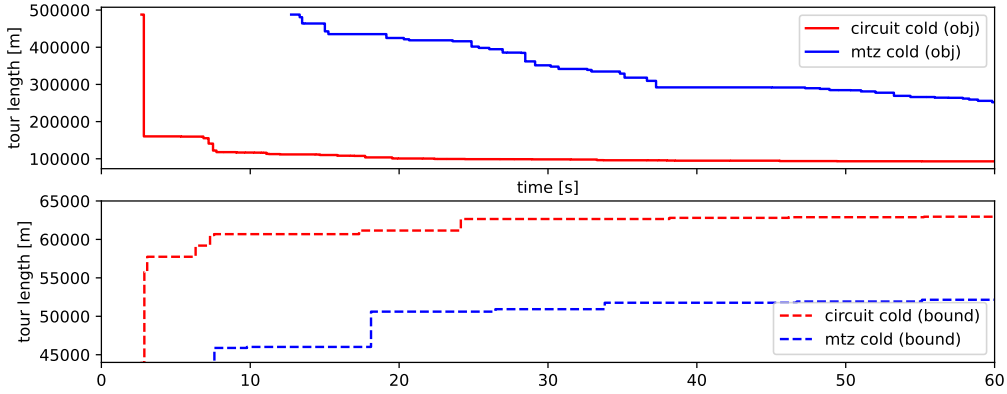
- MTZ has a very large (relaxed) polytope
- sometimes good with capacity constraints, order numbers, ...

Circuit Constraints (CP-SAT `add_circuit`)

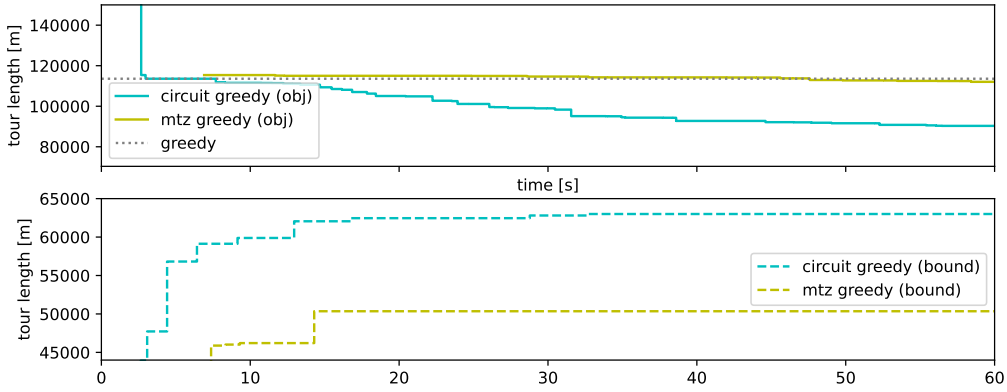
```python
def _circular_constraint(self):
  self.model.add_circuit([
    (self.node_id[u], self.node_id[v], var)
    for (u, v), var in self.arc_vars.items()
  ])
```

- high-level abstraction
- availability and performance depends on solver...
- ... if available often VERY good!

MTZ vs `add_circuit` (no warm-start)

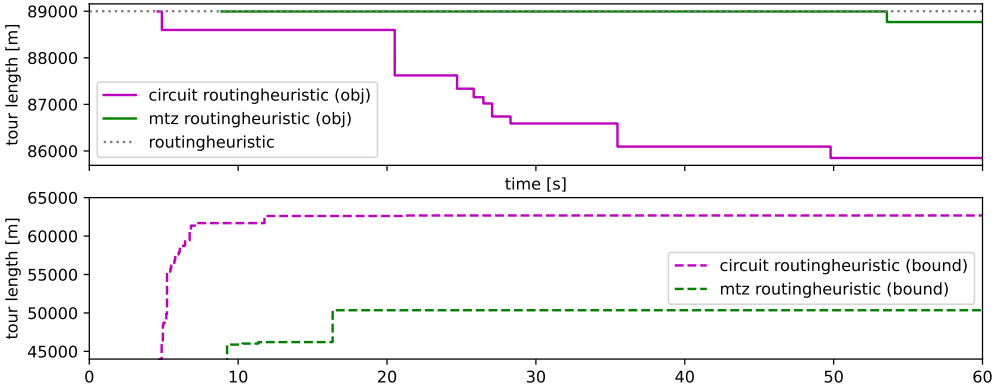


MTZ vs `add_circuit` (greedy warm-start)



a good heuristic can save a bad model, but cannot make a bad model good

MTZ vs `add_circuit` (specialized heuristic warm-start)



a good heuristic can save a bad model, but cannot make a bad model good