Group Projects

Deterministic Models and Optimisation

1 Branch-and-cut for the 0–1 Knapsack Problem

Consider the classical 0–1 Knapsack Problem and its Integer Programming formulation:

$$\max \quad \sum_{i=1}^{n} p_i x_i \tag{1}$$

$$\text{s.t.} \quad \sum_{i=1}^{n} w_i x_i \leq c \tag{2}$$

$$x_i \in \{0,1\} \quad \forall i \in \{1,\dots,n\}, \tag{3}$$

where we must maximise profits p_i collected packing n items of weights w_i in a knapsack of capacity c. For simplicity, we can assume that we deal with non-negative real values ($p_i, w_i, c \in \mathbb{R}_0^+$ for all $i \in \{1,\dots,n\}$) and we are not limited to integer values. We can also assume that the problem is not trivial, i.e., that $\sum_{i=1}^{n} w_i > c$.

Task 1 (2pt). Devise a simple and fast *primal* heuristic for the 0–1 Knapsack Problem. You have complete freedom on how to implement it, but note that it should run almost instantaneously on instances with up to 1000 items. You should not focus on finding the optimal solution within a long runtime but rather on finding a good solution in a short time. You can assume that you receive the items in the instance file already sorted by density (p_i/w_i).

Denote with N the set of item indices ($N = \{1,\dots,n\}$). A subset $C \subseteq N$ is called a *cover* if $\sum_{i\in C} w_i > c$, that is, the total weight of the items in the cover exceeds the capacity and, therefore, not all $|C|$ items in the cover can be packed. It follows that, given a cover C, inequality

$$\sum_{i\in C} x_i \leq |C| - 1 \tag{4}$$

is valid. Such an inequality is called a "cover inequality".

This inequality is not required for a correct model: (1)–(3) is complete and correct even without these inequalities. However, inequalities (4) can strengthen the linear relaxation of the model.

Task 2 (1pt). Devise a small instance in which the optimal solution of the linear relaxation of model (1)–(3) violates an inequality of type (4).

Next, we analyse a separation method for inequalities (4). Given a solution $x_1^*, \ldots, x_n^*$ of the linear relaxation of model (1)–(3), we want to find at least one cover C which yields a violated inequality of type (4), or prove that no such cover exists. Consider the following optimisation problem:

$$\min \quad \sum_{i=1}^{n} (1 - x_i^*) y_i \tag{5}$$

$$\text{s.t.} \quad \sum_{i=1}^{n} w_i y_i > c \tag{6}$$

$$y_i \in \{0, 1\} \quad \forall i \in \{1, \ldots, n\}, \tag{7}$$

which uses binary decision variables $y_i \in \{0, 1\}$ for each $i \in \{1, \ldots, n\}$. Note that x_i^* are *not* decision variables in this problem.

Question 3 (1pt). Is this a valid Integer Programme? Motivate your answer.

The problem defined by (5)–(7) is instrumental in devising a separation routine. In particular, if the objective value of the optimal solution to this problem is < 1, then a cover inequality is violated by solution $x_1^*, \ldots, x_n^*$. Otherwise, the solution satisfies all cover inequalities.

Task 4 (2p). Let $y_1^*, \ldots, y_n^*$ be the optimal solution of (5)–(7). Prove that if $\sum_{i=1}^{n} (1 - x_i^*) y_i^* < 1$, then there is a violated cover inequality. Explain how to build the corresponding cover C.

To practically solve the separation problem (5)–(7), we need one last observation.

Task 5 (1p). Prove that solving problem (5)–(7) is equivalent to solving a modified version of the 0–1 Knapsack Problem in which the capacity inequality is strict ($<$). Let us call this modified version the "01KP-Strict". In particular, given the original problem data and a solution $x_1^*, \ldots, x_n^*$, explain how to build the corresponding instance of 01KP-Strict which solves problem (5)–(7). Show how, in practice, an instance of 01KP-Strict can be turned into an instance of the classical 0–1 Knapsack Problem (the one with the $\leq$ capacity inequality) using a small numerical trick.

How interesting! We have a family of valid inequalities (the cover inequalities) for the 0–1 Knapsack Problem, and, to separate them, we can solve… another knapsack problem!

Note that any solution $y_1^*, \ldots, y_n^*$ which is feasible for (5)–(7) and whose objective value is strictly smaller than 1 gives a cover inequality. In other

words, we need not solve (5)–(7) to optimality and, therefore, we need not solve the corresponding equivalent knapsack problem to optimality. Therefore, now is the time for your heuristic algorithm from *Task 1* to shine.

Task 6 (3pt). Devise and implement a branch-and-cut algorithm for the 0–1 Knapsack Problem. Use Gurobi as the branch-and-bound solver and its callbacks for dynamic cut separation. The algorithm should start from the initial formulation (1)–(3), and every time the solver produces a fractional solution, it should attempt to separate violated cover inequalities. To do so, it will use your knapsack heuristic to solve the 0–1 Knapsack Problem associated with (5)–(7). There are two possible outcomes when you use your heuristic:

1. If it finds a solution corresponding to a violated cover inequality, you will add such a cut.
2. Otherwise, the ball is back in the solver's court. Because you have a heuristic solver, you cannot prove that there is no violated cover inequality. But, because cover inequalities are valid but *not* required, this is not an issue. We can tell the solver to keep exploring the branch-and-bound tree, and you will attempt to separate another cover inequality when the solver finds a new fractional solution.

Hint: Gurobi's preprocessing is quite aggressive and might close many instances before optimisation even starts. To give your B&C model a chance to prove its worth, disable Gurobi's preprocessing as follows:

```
model.Params.Cuts = 0
model.Params.Heuristics = 0
model.Params.Presolve = 0
```

You can generate your own (random) 0–1 Knapsack instances to test your algorithm.

2 Computational Experiments on the TSP

Recall the DFJ formulation of the Asymmetric Travelling Salesman Problem (ATSP):

$$\min \quad \sum_{i \in V} \sum_{j \in V \setminus \{i\}} c_{ij} x_{ij} \tag{8}$$

$$\text{s.t.} \quad \sum_{j \in V \setminus \{i\}} x_{ij} = 1 \qquad \forall i \in V \tag{9}$$

$$\sum_{j \in V \setminus \{i\}} x_{ji} = 1 \qquad \forall i \in V \tag{10}$$

$$\sum_{i \in S} \sum_{j \in V \setminus S} x_{ij} \geq 1 \qquad \forall S \subsetneq V,\ S \neq \emptyset \tag{11}$$

$$x_{ij} \in \{0, 1\} \qquad \forall i \in V,\ \forall j \in V \setminus \{i\}. \tag{12}$$

Your task is to perform an extensive computational analysis of the properties of this formulation on the popular TSPLIB instances. Here are the questions that I would like you to answer:

Task 1 (2pt). As we know, we can implement two types of branch-and-price algorithms to solve the above formulation. One separates violated subtour-elimination constraints (SEC) for integer solutions; the other performs separation for both integer and fractional solutions. Analyse the difference in performance between the two algorithms.

Task 2 (4pt). We cannot enumerate all SECs because there are exponentially many such constraints. However, we can enumerate upfront special subsets and add them to the initial formulation; the branch-and-cut algorithm will then add all the other violated SECs. For example, we can add upfront all the SECs related with sets of cardinality two ($|S| = 2$):

$$x_{ij} + x_{ji} \leq 1 \quad \forall i \in V,\ \forall j \in V \setminus \{i\}. \tag{13}$$

Or we could go one step further and also enumerate the SECs related with sets of cardinality three:

$$x_{ij} + x_{jk} + x_{ki} \leq 2 \quad \forall i \in V,\ \forall j \in V \setminus \{i\},\ \forall k \in V \setminus \{i, j\}. \tag{14}$$

On the one hand, we would make the initial continuous relaxation tighter, and we would avoid separating these constraints later. On the other hand, we would increase the size of the constraint matrix, potentially slowing down the solver. Analyse the impact of enumerating SECs for sets of size two (2-cycle elimination constraints), compared to the "vanilla" algorithm that does not enumerate any of them. Then, extend your analysis by also adding SECs for sets of size three (3-cycle elimination constraints). Are the results of your

analyses valid for both algorithms (integer-only and integer-and-continuous separation)?

Task 3 (4pt). A two-cycle elimination constraint for vertices i and j that are very far from each other is unlikely to be active because arcs (i, j) and (j, i) are almost certainly not used in a TSP solution. What happens if you enumerate k-cycle elimination constraints (i.e., SECs relative to sets of size up to k) only for close vertices? For each vertex i, consider its ℓ closest neighbours and only add k-cycle elimination constraints that involve these $\ell+1$ vertices. Try the following values for (k, ℓ): $(2, 3), (2, 4), (3, 4), (3, 5), (4, 5)$.

Suggestions:

- You can use a reduced subset of TSPLIB instances; otherwise, your computational experiments might take too long to run.
- Set a time limit for the Gurobi solver—for example, 15 or 30 minutes.
- Make sure that not all the instances you selected are trivial. There should be some that you cannot solve to optimality within the time limit.
- Think carefully about what metrics are interesting to report...you have complete freedom to decide how you want to communicate your results.
- Keep in mind that optimal solutions are known for the TSPLIB instances in case you need them for your analysis.

3 An extension of the TSP

Consider the problem of a repairman who gets called to fix home appliances. It is a booming moment for his business, and he receives more daily requests than he can take on. Being a clever repairman, he decides to use some Operational Research to decide on the jobs he accepts and refuses. He places all his customers on a directed loop-free graph $G = (V, A)$. The vertex set is $V = \{1, \ldots, n\}$; element 1 is the repairman's shop and elements $2, \ldots, n$ are customers. Each arc $(i, j) \in A$ is associated with a travel and service time t_{ij}, i.e., the sum of the time it takes to go from i to j, plus the estimated time he will spend at j. (The service time is zero for arcs that lead back to the shop.) Furthermore, each potential client $i \in \{2, \ldots, n\}$ has an associated profit p_i that the repairman will earn if he visits i. The workday lasts T units of time, and the repairman wants to maximise the profit he earns within this time limit.

He devises a model using the following variables:

- $y_i \in \{0, 1\}$ for each $i \in V$, taking value one if and only if the repairman visits location i.

- $x_{ij} \in \{0,1\}$ for each $i \in V$ and $j \in V \setminus \{i\}$, taking value one if and only if the repairman visits j immediately after i.

Then, he uses the following integer programming model:

$$\max \quad \sum_{i \in V \setminus \{1\}} p_i y_i \tag{15}$$

$$\text{subject to} \quad y_1 = 1 \tag{16}$$

$$\sum_{j \in V \setminus \{i\}} x_{ij} = y_i \qquad \forall i \in V \tag{17}$$

$$\sum_{j \in V \setminus \{i\}} x_{ji} = y_i \qquad \forall i \in V \tag{18}$$

$$\sum_{i \in V} \sum_{j \in V \setminus \{i\}} t_{ij} x_{ij} \leq T \tag{19}$$

$$\sum_{i \in S} \sum_{j \in V \setminus S} x_{ij} \geq y_k \qquad \forall S \subsetneq V\, (1 \notin S),\ \forall k \in S \tag{20}$$

$$x_{ij} \in \{0,1\} \qquad \forall i \in V,\ \forall j \in V \setminus \{i\} \tag{21}$$

$$y_i \in \{0,1\} \qquad \forall i \in V. \tag{22}$$

The objective function (15) maximises the collected profit. Constraint (16) makes sure that the tour includes his workshop, from where he starts and goes back. Constraints (17) and (18) ensure that the out- and in-degrees of visited customers are equal to one and those of non-visited customers are equal to zero. Constraint (19) limits the total amount of time spent outside travelling and servicing customer appliances. Finally, constraint (20) is a subtour elimination constraint analogous to the DFJ constraint in the Travelling Salesman Problem.

Task 1 (1pt). Explain why constraint (20) works as a subtour-elimination constraint.

There are exponentially many constraints (20) and, therefore, the repairman decides to implement a branch-and-cut algorithm to separate such constraints during the exploration of the branch-and-bound tree.

Task 2 (3pt). Devise and implement a separation procedure for (20) that works on integer solutions. Use Gurobi callbacks for the implementation.

Task 3 (3pt). Devise and implement a separation procedure for (20) that also works on fractional solutions. You can extend the separation procedure used for the Travelling Salesman Problem based on solving several max-flow problems.

Task 4 (3pt). Consider the procedure you devised in Task 3 and fix a given branch-and-bound iteration and its corresponding callback. Can your

procedure result in the same cut being generated twice within the same callback call? Of course, once added, the cut will not be violated any more and, therefore, it will not be generated again *in subsequent iterations*. But, *within the same callback call*, before the cut is added and the LP is solved again, the same cut may be generated more than once. There is no shame if this is the case; that's actually pretty common! If, after analysing your cut generation procedure, you establish that such a situation can happen, devise a way to avoid it. Otherwise, explain clearly why your procedure cannot generate the same cut twice.

You can generate your random instances to test the algorithm. Alternatively, you can use TSP instances from the TSPLIB, arbitrarily select one vertex as the repairman's workshop, attribute (random) profits to the other vertices, and fix a maximum total time that prevents you from visiting all clients.