Branch-and-Price for the Multiple Knapsack Problem : Implementation Report

Méthodes de recherche opérationnelle
IFT 6575

By

Zakary Gaillard-Duchassin

Presented to

Utsav Sadana

December 14, 2025

Université de Montréal

1 Introduction

This project implements the branch-and-price algorithm for the Multiple Knapsack Problem (MKP) described in Lalonde et al. [?]. The MKP asks : given n items with weights w_j and profits p_j, and m knapsacks with capacities c_i, how do we pack items into knapsacks to maximize total profit ?
The classic formulation looks like this :

$$\begin{aligned}
\max \quad & \sum_{i=1}^{m}\sum_{j=1}^{n} p_j x_{ij} && (1)\\
\text{s.t.} \quad & \sum_{j=1}^{n} w_j x_{ij} \leq c_i \quad \forall i && (2)\\
& \sum_{i=1}^{m} x_{ij} \leq 1 \quad \forall j && (3)\\
& x_{ij} \in \{0,1\} && (4)
\end{aligned}$$

where $x_{ij} = 1$ means item j goes in knapsack i.
The paper's approach is interesting because it reformulates the problem to separate the question of *which items to select* from *which bin they go in.* This turns out to make the algorithm much more efficient. My goal was to implement this algorithm in Java and see how well it works on the benchmark instances.

1.1 What I set out to do

— Implement the core branch-and-price algorithm from the paper
— Get it working on the SMALL benchmark dataset
— Compare my results with what they report in the paper
— Figure out what worked, what didn't, and what I learned

Full code is at `https://github.com/ZGaillard/BPMKP`. The README has build instructions and the Jupyter notebook `benchmark_analysis.ipynb` has all the result analysis and plots.
I want to do more runs until christmas, and try to fix the crash when running big instances. So please look at the github for updates.

2 The Algorithm

2.1 L2 reformulation

The key innovation in the paper is introducing new variables t_j that just say whether item j is selected or not :

$$\begin{aligned}
\max \quad & \sum_{j=1}^{n} p_j t_j && (5)\\
\text{s.t.} \quad & \sum_{j=1}^{n} w_j x_{ij} \leq c_i \quad \forall i && (6)\\
& t_j \leq \sum_{i=1}^{m} x_{ij} \quad \forall j && (7)\\
& \sum_{j=1}^{n} w_j t_j \leq \sum_{i=1}^{m} c_i && (8)\\
& t_j, x_{ij} \in \{0,1\} && (9)
\end{aligned}$$

Now the objective only depends on t_j, and the x_{ij} variables just handle the bin assignments. This means when we branch, we can branch on "is this item selected ?" instead of "which bin does this item go in ?", which dramatically cuts down the search space.

2.2 Column generation and patterns

The Dantzig-Wolfe reformulation [?] represents solutions as "patterns"—basically, feasible subsets of items that can fit in a knapsack. We have pattern pools P^0 (for the aggregated capacity) and P^i for each individual knapsack i.
The column generation works like this :

1. Start with a small set of patterns (empty patterns, single items, some greedy ones)
2. Solve the restricted master LP
3. Use the dual values to solve pricing subproblems (these are just knapsack problems)
4. If we find patterns with positive reduced cost, add them and repeat
5. Otherwise, we're done with this node

I'm using OR-Tools [?] for the LP solving (the GLOP solver).

2.3 Branching

When the LP solution has fractional t_j values, I pick the most fractional one and branch on it—create two child nodes, one where $t_j = 0$ (item excluded) and one where $t_j = 1$ (item selected).
One interesting case : if all t_j are integer but some x_{ij} are fractional, we know which items are selected but not how to assign them. In this case, I call a CP-SAT solver to try to find a feasible bin assignment. If it can't, I add a "no-good cut" that forbids this combination of items and resolve the LP at the same node.

2.4 What I didn't implement

A bunch of things from the paper that I skipped :

- No instance reduction preprocessing
- No MULKNAP warm-start
- The upper bound in constraint (31) is static—I don't update it with the incumbent
- No dual bound checks during column generation
- No stabilization or subgradient fallback
- Simple branching rule (most fractional) instead of their more sophisticated candidate selection
- CP-SAT only for packing checks—they use a three-stage approach (heuristic, then set-packing MIP, then arc-flow [?, ?])

So this is really a simplified version of their algorithm.

3 Implementation

3.1 Code structure

The code is in Java 25 with Maven. Main packages :

- `model/` and `io/` — data structures and file reading
- `formulation/` — the different MKP formulations
- `solver/lp/` — LP solver interface (wraps OR-Tools)
- `solver/cg/` — column generation
- `solver/bp/` — branch-and-price
- `solver/vsbpp/` — the packing feasibility checker

— `benchmark/` — batch benchmark runner

To run : `mvn compile` then `mvn exec:java -Dexec.mainClass=ca.udem.gaillarz.Main` for the CLI, or `MainBenchmark` for batch mode.

3.2 Design decisions

One thing I had to figure out : how to store pattern variables. Initially I tried using an `IdentityHashMap` but ran into problems because it uses reference equality, not content equality. So if I created two patterns with the same items, they'd be treated as different patterns.
I ended up making a `PatternVariable` class that wraps the pattern content, keeps track of which pool it belongs to, and stores the LP value. This made testing much easier since I could check pattern equality based on content.
For the VSBPP-SAT subproblem (checking if a set of items can be packed), I just use OR-Tools CP-SAT directly. The paper does something more elaborate with multiple stages, but CP-SAT works well enough for the SMALL instances.

4 Results

A more complete analysis of my results is given in the notebook `benchmark_analysis.ipynb`.

I ran the solver on the SMALL dataset (179 instances after one crashed) and a subset of FK_1. Configuration : 600 second time limit, 1000 node limit.

Dataset	Instances	Optimal	Gap limit	Avg gap	Avg time (s)
SMALL (1% gap)	179	105	74	0.000974	12.7
SMALL (0% gap)	179	179	0	0.0	16.5
FK_1 (10 inst., 0%)	10	10	0	0.0	8.5

With 0% gap tolerance (proven optimality), I solved all 179 SMALL instances in about 16.5 seconds average. If I allow a 1% gap, it's faster (12.7 seconds) but about 40% of instances stop at the gap limit instead of proving optimality.
The FK_1 subset also works well—all 10 solved quickly. I haven't tried the full FK_1 or the larger FK datasets yet (FK_2, FK_3, FK_4 would probably need the more sophisticated features I didn't implement).
All the detailed results are in `benchmark_results/` and the analysis notebook has plots showing runtime distributions, gaps by instance type, etc.

5 Comparison with the paper

The paper reports solving all 180 SMALL instances to optimality in 1.3 seconds average (though on a faster machine—about 2x faster than mine, so adjust to 2.8 seconds). My 16.5 seconds is slower, but I'm also missing several optimizations :

— No preprocessing to eliminate bins/items early

— Static upper bound weakens the formulation

— Simpler branching rule

— Less sophisticated packing solver

— OR-Tools instead of CPLEX (different numerics)

On the other hand, 16 seconds average for proven optimality on instances with 20-60 items and 10-20 knapsacks is pretty reasonable. The algorithm clearly works.
What I found interesting : allowing a 1% gap gives a 23% speedup but you'd need to look at the individual gaps to see if that's acceptable for your application. For some instances the gap stays tiny ; for others it might be the full 1%.

Other MKP algorithms include MULKNAP [?], 2D/PS+B [?], and the decomposition methods by Dell'Amico et al. [?]. The branch-and-price approach in Lalonde et al. [?] is current state-of-the-art for large instances.

6 What I learned

6.1 The reformulation really matters

The L2 reformulation with t_j variables is not just a theoretical nicety—it fundamentally changes how the algorithm explores the solution space. Branching on "item selected or not" instead of "item to which bin" is a huge win.

6.2 Pattern management is tricky

Getting the pattern storage right took some debugging. Java's reference vs. content equality for collections matters a lot. The explicit `PatternVariable` class was worth it for maintainability.

6.3 OR-Tools is powerful but has a learning curve

OR-Tools abstracts away a lot of details (basis management, tolerances, etc.) which is nice for getting started but makes debugging numerical issues harder. The CP-SAT solver works well for the packing subproblems though.

6.4 Column generation can be finicky

Some instances converged slowly. The paper's stabilization and subgradient techniques are probably necessary for harder instances. I saw some tailing-off behavior where the LP objective barely changes between iterations.

6.5 Implementation vs. theory gap

Reading the paper, it's not always obvious what's critical vs. what's optional. Turns out some of the features I skipped (like updating the constraint (31) bound) probably matter more than I thought. And the packing subproblem is harder than it looks—the paper's three-stage approach makes more sense after trying to solve these problems myself.

7 Future work

If I were to keep working on this :
First priority :

— Update constraint (31) with the incumbent value

— Add dual bound checks to column generation

— Try the three-stage VSBPP-SAT solver from the paper

Nice to have :

— Stabilization for column generation

— Better branching variable selection

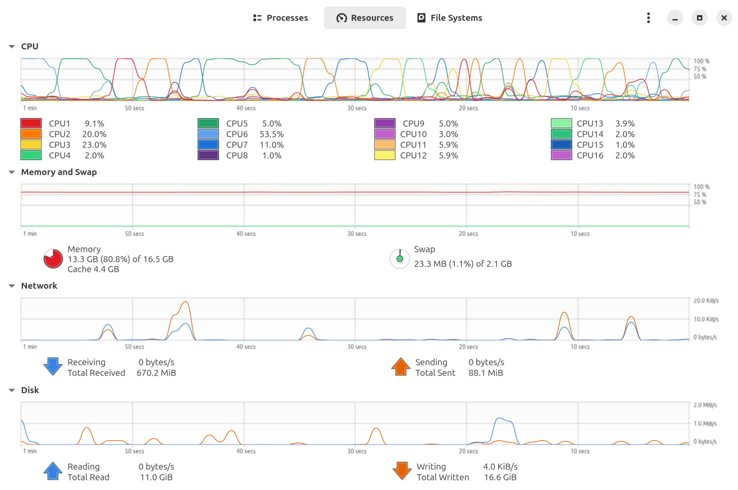
— Preprocessing and instance reduction

— Test on FK_2, FK_3, FK_4 (larger instances)

Overall though, this was a good learning experience. Branch-and-price is an elegant approach and the algorithm works pretty well even with my simplified implementation.

A System monitoring

For completeness, here's what system resource usage looked like during the benchmarks.
Figure 1 shows a single benchmark run—you can see the CPU and memory usage is pretty moderate, staying in one JVM process. Figure 2 shows three runs happening at once, which obviously uses more resources and they compete a bit.



🟥 CPU1	9.1%		🟩 CPU5	5.0%		🟪 CPU9	5.0%		🟩 CPU13	3.9%
🟧 CPU2	20.0%		🟦 CPU6	53.5%		🟪 CPU10	3.0%		🟩 CPU14	2.0%
🟨 CPU3	23.0%		🟦 CPU7	11.0%		🟧 CPU11	5.9%		🟦 CPU15	1.0%
🟩 CPU4	2.0%		🟪 CPU8	1.0%		🟨 CPU12	5.9%		🟪 CPU16	2.0%

Memory
13.3 GB (80.8%) of 16.5 GB
Cache 4.4 GB

Swap
23.3 MB (1.1%) of 2.1 GB

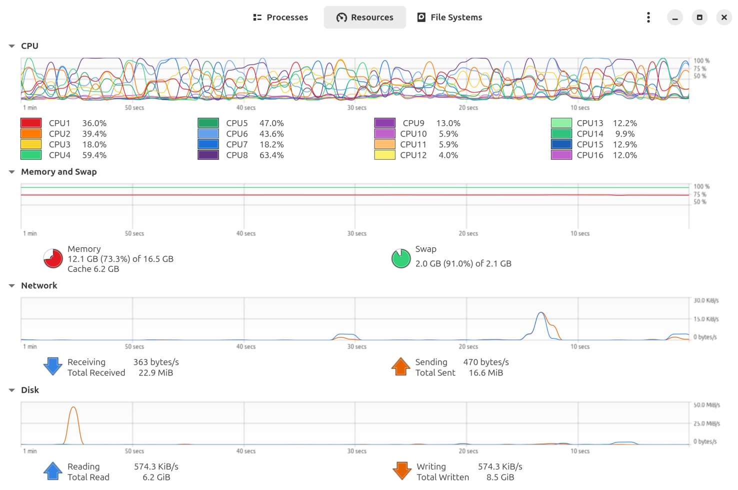
Receiving 0 bytes/s
Total Received 670.2 MiB

Sending 0 bytes/s
Total Sent 88.1 MiB

Reading 0 bytes/s
Total Read 11.0 GiB

Writing 4.0 KiB/s
Total Written 16.6 GiB

FIGURE 1 – Resource usage during one benchmark run.



🟥 CPU1	36.0%		🟩 CPU5	47.0%		🟪 CPU9	13.0%		🟩 CPU13	12.2%
🟧 CPU2	39.4%		🟦 CPU6	43.6%		🟪 CPU10	5.9%		🟩 CPU14	9.9%
🟨 CPU3	18.0%		🟦 CPU7	18.2%		🟧 CPU11	5.9%		🟦 CPU15	12.9%
🟩 CPU4	59.4%		🟪 CPU8	63.4%		🟨 CPU12	4.0%		🟪 CPU16	12.0%

Memory
12.1 GB (73.3%) of 16.5 GB
Cache 6.2 GB

Swap
2.0 GB (91.0%) of 2.1 GB

Receiving 363 bytes/s
Total Received 22.9 MiB

Sending 470 bytes/s
Total Sent 16.6 MiB

Reading 574.3 KiB/s
Total Read 6.2 GiB

Writing 574.3 KiB/s
Total Written 8.5 GiB

FIGURE 2 – Resource usage during three concurrent runs.

The memory requirements are reasonable for SMALL instances. For the larger FK datasets the process crash for now.