Jennifer Chen
Komron Aripov

Project – IV

jchen397, karipov
okeeffe, chernobyl

1 Model

Decision variables: We created a single vector of decision variables `use`, where for each of the available tests i, $\texttt{use}_i = 1$ if test i is used, and 0 otherwise.

Constraints: There must be at least one test that differentiates the two unique diseases. To implement this, for each pair of diseases j, k such that $j \neq k$, we ensure that $\sum_i^{\texttt{num\_tests}} \texttt{abs}(A_{ij} - A_{jk}) \cdot \texttt{use}_i \geq 1$ holds.

Objective value: Minimize the sum of the costs of the test, $\sum_i^{\texttt{num\_tests}} \texttt{use}_i \cdot \texttt{cost\_of\_test}_i$.

Our branch and bound algorithm architecture was standard and as discussed in lecture. We omit the BnB algorithm to discuss more interesting findings below.

2 Heuristics

2.1 Search Strategy

Best-first: This was our original implementation. We used a priority queue ordered by the LP objective value, popping off the minimum nodes first. This provided some good initial results, but we noticed that it took a very long time to find incumbent solutions, and many nodes weren't pruned, leading to a large queue and slow insertion times.

Depth-First: Instead of using a priority queue, we initialize a LIFO stack and pop off / insert at the end. This approach led to finding incumbents much quicker, and kept memory constraints reasonable. This simpler approach was our quickest.

Mixed: We also tried a strategy where the model begins with a depth-first search strategy, and once it finds an incumbent, it switches to a best-first search. We found inspiration from BnB algorithms that use best-first search but ocasionally "dive" down the search tree. However, this approach was almost equal to depth-first in time.

2.2 Variable Picking

Most fractional: Our first implementation chose variables based on their assignments at the parent node, choosing to branch on decision variables that had assignments farthest away from being integer. This proved to be a simple, quick and easy to implement heuristic that ended up being the fastest. Nonetheless, we went ahead and implemented more heuristics.

Random: We found that random variable picking was almost equally as efficient as any calculation-based variable choosing, which was surprising. We felt seen when we read an article by Morrison et al. (2016) that explained this phenomenon. We even attempted to do a local search by changing the numpy randomization seed to find the best seed for all the instances, and while seed 42 was very good, this heuristic was slightly slower than most fractional.

Random most fractional We thought to tone down the greedy approach and choose from a pool of top 10% most fractional variables. Still slower.

Cost-effective: We hand-made a problem-specific heuristic. We theorized that by taking the number of disease pairs which the test can distinguish and dividing this value by the cost of the test, we would include more effective tests in the solution set quicker. However, this made our model slower than any other heuristic – we're still unsure why.

2.3 Conclusions

For most instances given, it was true that any permutation of heuristics solves very quickly. What was necessary, then, was to optimize for very specific instances that were timing out, particularly `100_100_0.25_1.ip` and `100_200_0.5_5.ip`.

We deeply explored these and found that doing depth-first search with most fractional variable picking would find the most optimal incumbent node almost immediately, leading to lot of time gained, even at the cost of the smaller instances taking a second or two longer. The heuristics that are the fastest for our instances are almost certainly an "overfitting", and would probably not generalize well.

3 Further Optimizations

Double constraints bug: There must be some test that differentiates any two unique diseases. We initially added these constraints by double looping through pairs of diseases, but noticed this was adding constraints for pairs of diseases twice (both (k, j) and (j, k)). Only including unique disease pairs halved the number of constraints and also halved LP solve time. Easy to miss!

Threading and parallel branching: After noticing that for some instances the incumbent is found immediately and that most of the time is spent "verifying" the search tree, we were at a loss. However, we realized that each branch can be explored independently, and so we rewrote the codebase to use multiple threads that dispatch to CPLEX and each explore the search tree on their own, popping off the queue guarded by a mutex.

Profiling: Using a python profiler [1], we noticed that a large portion (roughly 20%) of time was spent simply initializing the model. We found an article by Beraudi (2023) that discussed ways to improve adding constraints, such as batching them.

Also in that article, we found that CPLEX automatically type-checks arguments passed to methods. Rewriting the code this in mind almost completely eliminated the time to initialize the model.

4 Overview & Time Spent

We pulled a couple all-nighters to work on this project, but not because we were late for the deadline. Changes made to the model were easily quantifiable, so when one experiment didn't work, there was an itch to try "just one more thing" before going to sleep. Together, we spent roughly 30 hours on implementation.

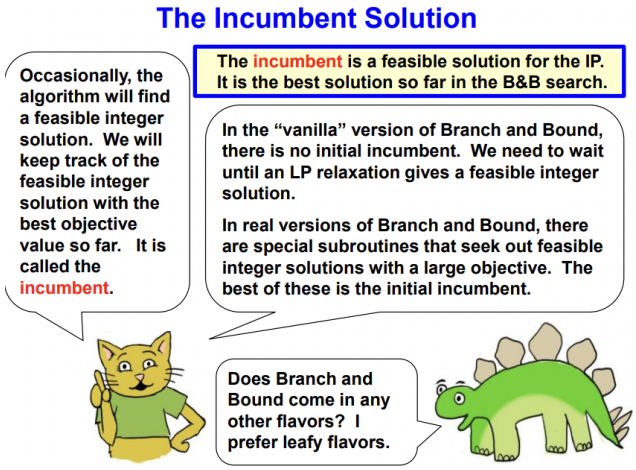
References

V. Beraudi. Writing efficient docplex code. https://github.com/IBMDecisionOptimization/docplex-examples/blob/master/examples/mp/jupyter/efficient.ipynb, 2023. This article was very helpful in understanding common ways of increasing CPLEX model efficiency in Python. Accessed: 2024-04-12.

David R. Morrison et al. Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning. *Journal of Global Optimization*, 64(2):210–227, 2016. doi: 10.1016/j.disopt.2016.01.002. URL https://www.sciencedirect.com/science/article/pii/S1572528616000062. Referring to section 4.2, "However ... [the most-fractional] branching rule is in general no better than selecting a branching variable at random in terms of computational

time required and number of subproblems explored." Accessed: 2024-04-12.

J. Orlin and E. Nasrabadi. Integer programming: branch and bound, 2013. URL https://ocw.mit.edu/courses/15-053-optimization-methods-in-management-science-spring-2013/resources/mit15_053s13_lec12/. This lecture helped us to conceptually understand the Branch and Bound Algorithm. We also love the cartoons.



5 Addendum

```
77.356 <module>  ../../src/main.py:1
├─ 76.343 main  ../../src/main.py:7
│  ├─ 55.150 IPInstance.solve_ip  ../../src/ipinstance.py:264
│  │  └─ 55.149 IPInstance.branch  ../../src/ipinstance.py:140
│  │     ├─ 53.621 IPInstance.solve_lp  ../../src/ipinstance.py:252
│  │     │  └─ 53.621 Model.solve  docplex/mp/model.py:4844
│  │     │        [14 frames hidden]  docplex, collections
│  │     │           50.679 CplexEngine.solve  docplex/mp/cplex_engine.py:1983
│  │     │           └─ 50.670 Cplex.solve  ../../../../../../local/projects/cplex/CPLEX_Studio2211/cplex
│  │     │              └─ 50.667 lpopt  ../../../../../../local/projects/cplex/CPLEX_Studio2211/cplex/py
│  │     │                 └─ 50.665 CPXXlpopt  ../../../../../../local/projects/cplex/CPLEX_Studio2211/
│  │     │                    └─ 50.336 [self]  ../../../../../../local/projects/cplex/CPLEX_Studio2211/
│  │     └─ 1.462 IPInstance.remove_decisions  ../../src/ipinstance.py:248
│  │        └─ 1.462 Model.remove_constraints  docplex/mp/model.py:3694
│  │              [3 frames hidden]  docplex
│  └─ 21.193 IPInstance.__init__  ../../src/ipinstance.py:67
│     ├─ 11.252 <listcomp>  ../../src/ipinstance.py:91
│     │  ├─ 8.502 Var.__rmul__  docplex/mp/dvar.py:280
│     │  │     [11 frames hidden]  docplex
│     │  └─ 2.749 [self]  ../../src/ipinstance.py
│     ├─ 4.347 Model.sum  docplex/mp/model.py:3329
│     │     [6 frames hidden]  docplex
│     ├─ 2.883 <listcomp>  ../../src/ipinstance.py:90
│     └─ 0.784 Model.add_constraints  docplex/mp/model.py:4155
│           [2 frames hidden]  docplex
└─ 1.004 <module>  ../../src/ipinstance.py:1
```

Figure 1: Python profiler shows `__init__` takes a whopping 20 seconds to instantiate a model.

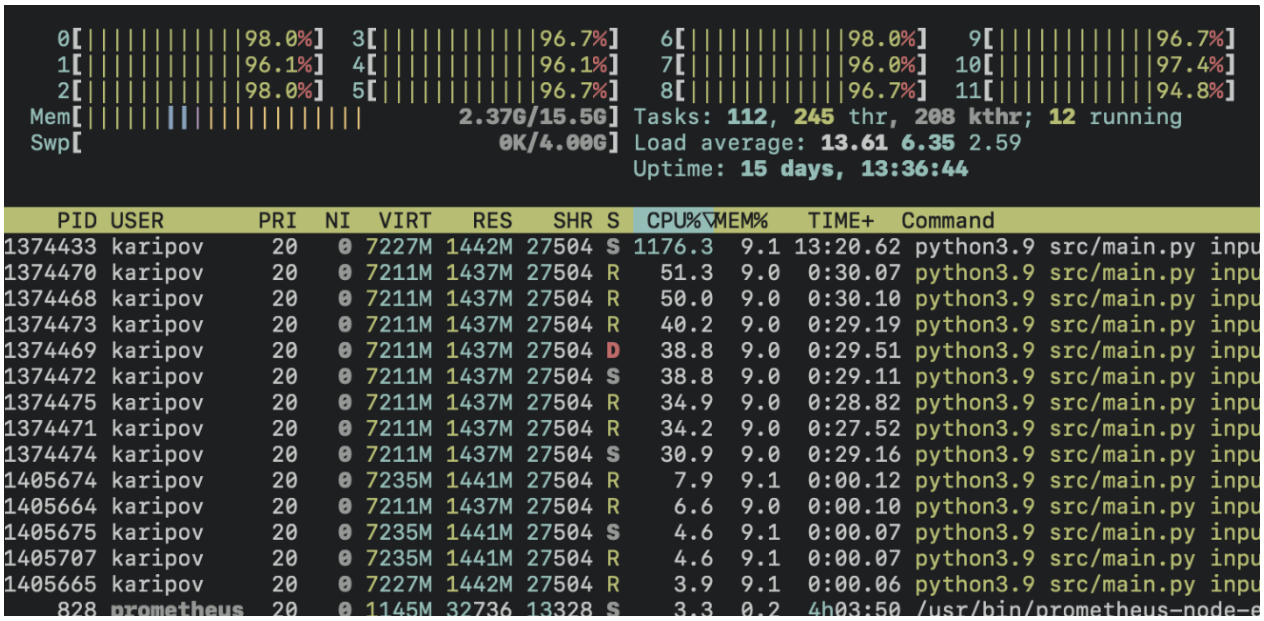


Figure 2: Twelve cores running twelve threads that solve LPs in parallel.