Constraint Model

The situation we are asked to model is related to using symptoms to diagnose disease. Specifically, we are asked to find which tests we need to isolate what disease a possible patient might have. If we model this problem as a matrix of tests by diseases, essentially we need each column to be unique. By that, I mean that each disease should have a unique result when all selected tests are used, such that we know what disease the patient has. This would be difficult to express, even if we had higher level constraints, so we perform a simplification of the problem. We create a difference matrix of numTests $\times \binom{\text{numDiseases}}{2}$. For every possible pair of diseases, we compute the XOR of the test results. This expresses whether a test can be used to tell the difference between the two diseases.

Decision Variables

The decision variables are simply a binary array, representing if a test is used or not. For this problem, although it is an integer programming problem, we perform a linear relaxation, so we make the binary array continuous.

Constraints & Objective

We create a constraint for each column in our difference matrix. For each column, we multiply every value by whether the test is used, then ensure that the sum of the products is greater than or equal to one. This represents that for every pair, there is at least one test that can tell them apart. This is the reason why we used XOR, as it tells us if the test has different results for the pair of diseases it corresponds to. The objective of this problem was to minimize the total cost. We calculate the total cost by multiplying the cost of a test with whether that test was used or not.

Branch and Bound Algorithm

Progress on this algorithm was difficult, as I coded multiple versions of it and to some extent I felt like I was essentially doing grid search. I also initially coded the entire project in Python, but my results were extremely slow and I decided to switch over to Java. With the same implementation in Java, I saw a marked improvement. The logic of the algorithm seemed intuitive, using a linear solution and then at each branching opportunity adding a constraint to push the previous solution to a new linear solution that would hopefully also be an integer solution.

Heuristics

The heuristic pertains to which test I decided to split on in the intermediate linear solutions. I explored using greatest fractional, first available, and closest to 0.5 for my heuristics. First available did not end up being as quick as the other approaches but closest to 0.5 as well as greatest fractional ended up having similar runtimes. I theorize that the reason for this would be because regardless of whether a test is fractionally used or exactly half used, it means that the test is only partially

used and so needs to be constrained anyways. I assume that greatest fractional or least fractional would be more useful for variables that are not binary, because they are solutions that are almost integer solutions, but require slight modification.

Search Methods

I used best first search, recursive depth first search, and a hybrid iterative depth first search. Best first search yielded the best results, followed by recursive depth first search, and last was the hybrid iterative search. Best first search was always superior to the other searches, but it yielded a slight speed up when I made the following modification. In best first search, an integer solution is unlikely to arise until a significant period through the search, as the best option is likely a solution that has less constraints. Since all variables in this question are meant to be binary, I simply rounded them up to compute an upper bound. In the linear solution if the test's value was not zero, then I forced it to be one and calculated this cost. I constantly updated the upper bound with this value, so I was able to prune the tree in an effective manner. I expected it would have made more of a difference, but I theorize that the parts of the tree that are meant to be pruned are essentially sent to the back of the priority queue and are only explored as a last resort.

Parallelization

Despite these improvements, I saw that my implementation was still slow. To remedy this, I tried to parallelize this process of popping off the queue and finding new linear solutions. This resulted in a significant speed up, as every time I pop off the priorirty queue, I needed to find a variable to branch on, I had to add all corresponding constraints, solve the linearly relaxed model, then add this solution and its constraints to the priority queue. I ended up using four threads, which almost halved the time for the longer problems. I experimented with more threads, but it did not provide any speed up, likely because of overhead but also because there is not much point in exploring the eight best solution in most cases. I found that there was slowdown when I recreated the model from scratch every time, so at every node, I added all the necessary constraints then removed them, so I did not have to recreate the model and the constraints that were common to every model did not have to be modified.

Conclusion

Overall, this project was quite satisfying. I enjoyed the two part nature of this project, where it was both an engineering problem and a modeling problem. I was able to incrementally work on this project, from thinking of a high level constraint model to making it an integer model and also starting with a rudimentary branch and bound algorithm and then trying different things to speed it up. The necessity to change languages was a little bit frustrating and I have a feeling that the fastest implementations will likely be striking a balance between runtime of the Branch and Bound algorithm and compatibility with IBM CPLEX. I even profiled some of my work in Python and I was very confused to see that certain CPLEX functions, like cloning models was so slow compared to adding and removing many many constraints.