1 Constraint Model

In this section, we describe the constraint model we use for the LP solver that computes the bounds used in the branch-and-bound algorithm. Let $x_{i,(J,K)} \in [0,1]$ determine if the i_{th} test differentiates between the pair of diseases, (J,K). Then, for each disease pair (J,K), we impose the constraint: $\sum_i^{numTests} x_{i,(J,K)} \geq 1$. This results in a total of $n \cdot (n-1)/2$ constraints, where n is the number of diseases.

2 Branch-and-Bound

In this section, we describe the structure of our branch and bound algorithm:

2.1 Search

Our initial solver proceeded in a depth-first fashion, but upon switching to best-firs search, we observed that best-first search yielded a performance increase. To implement best-first search, we utilize a heap to order the nodes of our search tree.

We initially define the heap's comparator solely based on the objective value of the nodes. If a given node of our search tree has a worse objective value, we will search it after any node with a better objective value.

One benefit of this search architecture is that as soon as we pop a node from the heap that has a worse objective value than our incumbent, we can drop the heap completely, as we know the remainder of the nodes on the heap are all prunable. However, we observed in some problem instances that under this schema, the solver took quite a long time to reach an incumbent. To experiment, we tried modifying the heap's comparator to add noise to the objective value and also weight deeper nodes (i.e. nodes with more fixed variables) to be preferential in order to encourage the solver to prioritize nodes already close to an integral solution.

2.2 Heuristic

We tried a lot of different branching heuristics for this project, but what we found worked best in the end was just branching on the variable whose assignment in the most recent solve of the LP relaxation was closest to 1.0. This indicates tests which the LP solver found very useful for its cost, so it would intuitively be a good variable to fix and see how that affects the objective function. With B(est)FS, we don't really have to decide which value to fix the branch variable to first, as we can just fix it to 0 and to 1, solving an LP relaxation for each. Then, we'll revisit those nodes in the order of how good their objective values are.

2.3 Multi-Threading

Our final solver uses a multi-threaded approach which uses a single *manager thread* and then $n-1$ *worker threads* (where n is the number of cores on the machine). At a high level, the manager handles the global state for the search, including a binary heap to store active nodes in the search,

the current incumbent, and a structure that records relevant information for analytics. Workers, on the other hand, have a copy of all the constraints for the LP relaxation, and their job is to visit nodes in the BnB tree and determine what to do next as quickly as possible.

The manager thread continually removes the best active node from its heap according to our search heuristic and enqueues that node onto a work queue. Workers are continually de-queuing from the work queue, so they will see that a node has been added. Then, the worker can make a branching decision at that node and solve both LP relaxations resulting from the branch. After solving each relaxation, the workers send their results back to the manager (through a channel) for decision-making. If the solve's solution was integral, the manager will conditionally update the incumbent depending on how good that solution was. If not, we can prune doing further work on the node if the relaxation was infeasible, or if its solution's objective value was worse than our current incumbent. Otherwise, more work must be done on this subtree, so we add this node back into the heap for further work in the future.

2.4 Programming Language

We engineered a proof-of-concept in Python, but ultimately transitioned to Rust. One interesting thing we observed in Python, was that setting the model's thread paramter to 1 greatly improved the performance of the LP solver.

Moving to Rust was pretty annoying. We ended up using a modified version of an open-source crate for CPLEX bindings in Rust to access CPLEX. However, we did observe for some simple experiments, (e.g. calling LP solve on one of the problem instances 1000 times) performed faster on Rust compared to Python.

2.5 Analytics

To accurately measure the performance of our solver and how different heuristics affected it, we added some extra instrumentation for quickly recording its performance statistics. For example, each worker thread recorded the total number of LP solves it had to do, the total amount of time spent waiting to receive orders, and the total amount of time it spent solving LP relaxations. Similarly the manager thread recorded global information like the total number of solves by any worker, the maximum heap size, and the outcome of each node visited in the search tree (e.g. whether it was pruned, infeasible, or resulted in an integral solution, or another active node that had to be searched). These were super useful for profiling different heuristics & search strategies to ensure they were resulting in a lot of pruned subtrees or good integral solutions.

3 Time Spent

Roughly around 20 hours each.