I. Data Structure and Algorithm:

1. CirMgr:

   A GateList with size of total gate number is used to store all PI gates and AIG gates. The pointer of each gate is stored in the slot corresponding to their IDs, so that accessing gates can be constant time. PO gates are stored in another GateList.

   For each gate, the ID of fanin gates are stored, while the CirGate pointer of fanout gate is stored. This storing scheme is convenient for me to manipulate the inverting of fanin gates.

   The DFS list is also stored in CirMgr as a GateList since it is heavily used in most of our functions.

2. Strash:

   The purpose of this function is to merge two gates with the same fanins. When two gates A and B are merged together, the following three steps are executed.

   First, B’s fanout gates shall change their input fanin IDs to the literal ID of A based on the sign of their connection. Second, B’s fanin gates shall delete their fanout to B and reconnect to A. Lastly, B’s fanout gates shall be pushed into A’s fanout list.

   In this function, a HashMap is used to store the keys and the CirGate pointers. Since there are two fanins, they are multiplied by huge prime numbers respectively and later summed together to form a key. This calculation can be done in constant time, and the key value is guaranteed to be much larger than the hashmap bucket size. The performance of this function is quite satisfying, so that this scheme may effectively reduce the number of collisions.

3. Simulate:

   Parallel pattern encoding is used in this function. 32 patterns are stored inside an unsigned integer, and then we can use bitwise operation to calculate the simulation values for all gates. A HashMap is used to store the <Simulation key, CirGate*> pairs. However, the insert function is different from the optimize function version; it won’t allow collision to happen, and when a new node with different key value needs to be inserted, it will be stored into a separate vector instead. After that FEC group is simulated, the vector with collision gates will compare the key values of every possible pair to collect potentially equivalent pairs. The
HashMap can ensure linear complexity, yet the handling of the collision vector can be of $O(n^2)$ complexity. Although collision won’t happen that often, it is still sometimes quite time consuming.

Since we need to identify FEC pairs and IFEC pairs as well, it will be nice to offer them the same key. In the key generation function, the MSB of the simulation value is checked; if it is 1, the whole simulation pattern is inverted to form the key; otherwise, the original simulation value will be the key. The HashSize is determined by the getHashSize() function.

For the random simulation stopping criteria, if the FEC group size doesn’t change after a determined maxFail value, the random simulation is stopped. The maximum fail value is estimated as square root of the total gate number divided by a constant. Since parallel pattern is used in simulation and fail counter will only increase by 1 after 32 patterns, the constant is chosen as a large number to accelerate simulation.

4. Fraig:

The SAT solver is called to determine if the FEC pair is functionally equivalent. If the given xor function is reported to be unsatisfiable, one of them will be merged. The whole FEC group set is called repetitively until the group size doesn’t change anymore. Yet, this consumes a lot of memory and runtime, so the whole stack might overflow and shut down the fraig function.

At first, I have tried to utilize the SAT results to discriminate potentially different pairs, but there isn’t enough time for me to finish it. The basic idea is illustrated as follows:

If one of the FEC pairs is reported as satisfiable by the solver, it is highly possible that this input pattern can also identify other FEC pairs within the same group. Thus, when a SAT result is reported, every remaining pair in this group will be classified into two groups based on this pattern. If any of the groups has multiple gates, a new FEC group is created, and the original one is deleted. This trick can divide huge FEC groups into smaller ones, and therefore improve the overall performance.
II. Analysis of experiments:

For the basic functions such as sweep(), strash(), and optimize(), the output and error message of my function is fine tuned to exactly match the reference program. Runtime is compatible, yet the memory usage is slightly higher than the reference program.

My Simulation program works fine on typical circuits, but it can be significantly slower than the reference program when large circuits such as sim13.aag is optimized. I believe a better data structure to handle collision gates can be really helpful in improving its efficiency.

My fraig function can only be done on smaller circuits, and it will consume too much memory and lead to stack overflow when running on larger circuits.

1. Experiment with run.opt

The script is modified to do sweep(), optimize(), and strash() once. It is performed on the rather small opt 01~07 circuits. Both the reference program and my program have runtime 0 second. But the average memory usage is slightly different.

<table>
<thead>
<tr>
<th></th>
<th>My Program</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory usage</td>
<td>0.1758M Bytes</td>
<td>0.1172M Bytes</td>
</tr>
</tbody>
</table>

I guess the reason of my excess memory usage results from some flags and DFSList stored inside the CirMgr for convenience. Also, a lot of space is wasted for storing the PI and AIG gates, since they are stored in corresponding ID slots and this vector may be really sparse when the circuit is further optimized.

2. Experiment of basic functions on simXX.aag

In this experiment, function performance on larger circuits are estimated.

I created a script that calls the three functions in the mentioned order for five runs to ensure the circuit is completely optimized before simulation and fraig. All sim 01~15 files are performed, while only a few of them with larger difference is reported here.

<table>
<thead>
<tr>
<th></th>
<th>My Program</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sim12.aag</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The runtime is quite comparable, yet the memory usage can be around twice as large as the reference program. Sweep and optimize function doesn’t need too much extra memory to perform, so I guess the reference program has a better control on the hashmap size, and maybe even a better key.

3. Experiment on file simulation

The run.fsim script is used to estimate the performance of the file simulation. As the table, you can see that the runtime and memory usage can grow really high on some of the gates. I have heard that a binary search tree implementation for storing FEC groups will be a lot faster and memory efficient, yet I don’t really have time to modify my code and give it a try.

4. Experiment on random simulation

Similar as the above experiment, the run.rsim script is used to estimate the performance of my random simulation function. For smaller circuits, the stopping criteria is quite good and the runtime is quite satisfying; however, with larger circuits, the max fail counts may be too large and may take too long to run the whole circuit. Also, my program may not be able to identify all FEC pairs, and this may make the burden of the fraig function even heavier.
5. Experiment on fraig function:

The run.fraig script is used to check performance of my fraig function.

The runtime may be significantly higher, and sometimes the output circuit may be wrong. Yet, in smaller circuits, it seems quite correct.