姓名：施順耀
學號：B03901133
聯絡資訊：b03901133@ntu.edu.tw / 0939468169

資料結構 Final Report

a) Design for Simulation

Simulation 主要分成三個部分：

1. 取得要拿來Simulate的值
2. Simulate
3. Collect FEC groups

一之一、取得 Simulate 的值

我在每個 gate 中都存入一個 simValue ，型態為unsigned long long，用來記錄每個 gate 目前所存的值。因此，在 randomSim 中，我利用了 “util.h” 中的 RandomNumGen 給我兩個 32 bits 的數字，再將這兩個數字，塞叫一個 unsigned long long 裡，再餵給PiGate，而在 fileSim 中則是將file中的數字 string 直接轉換成unsigned long long ，再餵給 PiGate。

一之二、設定Random Simulate 給定pattern數目的想法

RandomSimulate 設定的pattern數，在當netlist.size() < 100 時，在直接給訂netlist.size() 的pattern數，而超過100時，則給根號netlist.size()，設定門檻的原因在於，如果今天給訂的電路gate數量不多，simulate所要花的時間並不多，而simulate又能有效的分fec groups ，而能減輕fraig的負擔，但是當netlist的大小有一定數量時，就不能這麼做了。

原因在於，若是今天給訂的電路大小約為 N ，而 All-gate simulate的方法執行一次的時間也約為 N ，若在random simulate時，就做了 N 次的 simulate，則總共大約就需要 N^2 的時間，這樣的時間複雜度，差不多相等於後面真正簡化電路的 Fraig 的N^2 ，如此一來便失去simulate增加簡化電路的速度的目的了，因此最後決定採用大約根號N 的pattern數。

二、Simulate

我在simulate中的設計十分直接，主要就是將DFSlist 中所有的 gate 去呼叫對應的 simulate() ，由於依照 DFS 的順序，能保證每個 gate 在自己 simulate() 前，其 fanin->simulate() 都已經被呼叫過。

三、Collect FEC groups

這個部分主要分為兩個階段，第一個階段是對於目前的 fecGrpList 中的每個 fecGrp 中的每個 gate ，利用 HashMap 對於當前不同的 simValue 進行分類，而 HashMap 的Size 則取決的每個 fecGrp 的大小，

在 HashKey 的設計上，我傳進每個 gate 的 simValue ，並且固定讓最後一位 = 0，以確保 FEC 和 IFEC 會被存到同一個 fecGrp。

這個階段主要的演算法對於同一個 fecGrp 的每個 gate 去對 HashMap 做 check，如果已經 Map 中已經有了，則將這個 gate 加進已經在 Map 中的那個 gate 所在的 fecGrp ，若沒有，則以這個 gate ，創造一個新的 fecGrp。

第二階段則是蒐集有效的 fecGrp，在前一階段的演算法中，有可能在 fecGrpList 中留下只有一個 gate 的 fecGrp，另一方面，也必須刪除已經處理過的 fecGrp，這部分的演算法，我沒有採用vector.erase()，原因在於每次的 erase 都必須花掉 O(n)的時間，取而代之的是，我建立一個新的 vector< fecGrp> ，然後對於每個在原本fecGrpList的 fecGrp 都去看他的 size 是否大於1 ，若大於，則將這個fecGrp 丟進新開的 vector< fecGrp >，最後再以這個 vector< fecGrp > 覆蓋掉原本的 fecGrpList ，如此便完成一次的identify FEC groups。

b) Design for Fraig

對於Fraig的加速，我主要分成三個部分，而 Fraig 最基本的演算法就是對於所有的 fecGrp 兩兩去看裡面的 gate 藉由SatSolver 證明是否真的是Fec。

分成三個部分：

1. sortFecGrpList()
2. resimulate()
3. 捨棄很可能不一樣的fecGrp

一、sortFecGrpList()

如果今天我們直接從fecGrpList中的第一個 Grp做到最後一個，很可能遇到需要證明很久的 Grp，因此sortFecGrpList() 是在嘗試將fecGrpList 依照期待的證明難易度進行 sort，而期待的證明難易度估計方式簡單來說是看每個 gate 走到 Pi 的最遠距離，實際作法為：

```
distances of Pi, Const gate = 0
for each Aig gate in netlist
    if ( distance of the first faninGate > distance of the second faninGate )
        distance of this gate = distance of the first faninGate + 1;
    else
        distance of this gate = distance of the second faninGate + 1;
sort fecGrpList by the sum of distances of each gate in each fecGrp
```

如此便能一定程度的將fecGrp排序，而前面容易證明的 pair 所累積下來的 pattern 也有助於分析後面較難證明的 fecGrp。

二、resimulate()

當我們每次證明某個pair 並非相等時，可以從 SatSolver 拿到一組使得這兩個 gate 不相等的 input pattern ，而這個 pattern 比起random 產生的pattern 更有機會將其他fecGrp 分開，因此在fraig 中，我會存下每個不相等的 pattern，而累積到某個數字時，便會丟給simulate ，試圖分出不同的 fecGrp，而這個上限主要取決於 dfsList 的大小，主要是當 dfsList的大小小於100時，就直接等於其大小，若大於，則取根號dfsList的大小除以10。

三、捨棄很可能不一樣的fecGrp

有時會有很可能都不一樣的fecGrp會佔據不少 fraig 的時間，而我也設計了一個機制，使得程式不會一直卡在某一個 fecGrp ，每當一個fecGrp 開始證明，便會有個計數器，每當有一個 pair 被證明為不相等時，便會 +1 ，而當這個計數器到達 這個fecGrp大小的一半時，便會捨棄剩下還沒證明完 fec pairs。

c) Experiments

實驗－：RandomSimulate 次數的多寡

目的：先猜測randomsimulate的次數對於fraig電路的最佳優化，再以實驗微調參數，期望增加fraig的執行速度。

實驗流程：先以原先的方法執行cirsimulate -r; cirfraig; 記錄執行時間後，再調整參數，並觀察變化。實驗的例子會以能化簡成 Const0中最複雜的電路 sim12.aag 以及雖然不能化簡成 Const0 中最複雜的電路 sim13.aag 作為時間參考，而simulate 的pattern ，則是 pattern.12 以及 pattern.13。

實驗預期：random simulate 給的pattern數目應該可以比 根號N 再少一些，原因在於在fraig時還會再蒐集更有用的pattern拿去resimulate，因此在random simulate的時候不需要給太多的pattern，以免浪費時間。

實驗二之－：三種Fraig 加速方法比較

目的：觀察當加上三種方法任一種時，會對整體的執行時間以及所剩電路造成多少影響。

實驗流程：實驗的例子如同實驗一，並分別加上三種優化方法，比較時間以及最後結果。

實驗預期：若是只有sortFecGrpList，最終要做完的數量還是固定的，預期對於執行時間沒有太大影響，而若能加上resimulate()，對於時間的幫助應該蠻大，且剩餘的gate數量應該是一樣的，而單純捨棄很可能不一樣的fecGrp，預期會大大增加速度，但若是沒有其他配套措施，很可能剩下很多gate。

實驗結果與討論：

原始時間（沒有任何Optimize）：

sim12.aag：

剩餘gate數：5736　　執行時間：16.12s
sim13.aag：
剩餘gate數：無法估計 執行時間：非常久...

加上sortFecGrpList()：

sim12.aag：
剩餘gate數：5736　　執行時間：14.21s
sim13.aag：
剩餘gate數：無法估計 執行時間：非常久...

加上resimulate()：

sim12.aag：
剩餘gate數：6116　　執行時間：8.39s
sim13.aag：
剩餘gate數：78968　　執行時間：74.28s...

捨棄很可能不一樣的fecGrp：

sim12.aag：
剩餘gate數：5795　　執行時間：8.94s
sim13.aag：
剩餘gate數：79016　　執行時間：143.8s
ref: 剩餘gate數： 5764個/sim12.aag　　78875個/sim13.aag

從結果來看，沒有任何優化的fraig 在sim13.aag 的優化上，時間非常的長，而sortFecGrpList() 也如預期對於執行時間沒有很大的幫助，但這方面需要再進一步的實驗。

resimulate() 如預期對於執行時間有大大的優化，以sim13.aag來看，無論是剩餘的gate數量或是執行時間，都是最佳的，但sim12.aag 的剩餘gate數量有些異常。

捨棄很可能不一樣的fecGrp對於執行時間也算有很大的優化，且在sim12.aag 捨棄的gate數量也不多，但sim13.aag 雖然執行時間有很大的進展，但捨棄的gate 數量有點多。

實驗二之二：三種Fraig 加速方法比較

目的：由於在實驗一之一中，發現若是單純只有sortFecGrpList()，似乎沒有什麼幫助，且若單純捨棄很可能不一樣的fecGrp，也很可能造成剩餘的gate數量過多，因此在這個實驗中會比較同時有兩種優化（sort 和 resimulate）以及同時有三種優化的方法。
實驗流程：同實驗二之一

實驗預期：相比只有sortFecGrp 同時有兩種優化應該表現較佳，而若同時有三種優化應該能達到最好的執行速度。
實驗結果與討論：

sortFecGrpList & resimulate：

sim12.aag：
剩餘gate數：5806　　執行時間：6.34s
sim13.aag：
剩餘gate數：78942　　執行時間：88.43s...

sortFecGrpList & resimulate & 捨棄fecGrp：

sim12.aag：
剩餘gate數：5824　　執行時間：5.56s
sim13.aag：
剩餘gate數：78973　　執行時間：74.53...

同時有三種優化確實執行時間快了許多，雖然剩餘gate的數量較多，但在可接受的範圍，而最後我的fraig() ，也同時採用了這三種的優化。

實驗三：Fraig 中的常數測試

目的：期望藉由調整 Fraig 中各項優化的參數，期望達到小幅度的速度增長。
實驗流程：主要調整resimulate的次數門檻以及在同一個fecGrp中遇到幾次證明不相等的pair 要放棄整個fecGrp。
實驗預期：預期理論上比起根號netList的大小，resimulate 的次數應該可以再往下微調，而放棄的次數門檻理論上越低，執行時間能夠越快，然而，剩餘沒證明完的gate數量卻會增多。
實驗結果與討論：經過調整與測試後，resimulate的門檻大約在根號除以十的大小左右時，能夠有最佳的performance，而放棄的門檻，約在某一個fecGrp的大小的一半左右。

d) 瓶頸及缺點的理論探討

Simulate：

由於在simulate的部分，我使用了all-gate的simulation的方法，若是拿到的pattern 相鄰兩個pattern 極為相似或是根本長得一樣時，必須再花一次N的時間，然而，若是用遞回的方式進行simulate，則可以在很快的時間內完成，但相對的若是遞回時都沒有發現跟上一個pattern一樣的output，就會花比較多的時間Handle。

Fraig:

Fraig部分的瓶頸很可能在resimulate的時機點，以及放棄整個grp的時機，假想今天的fecGrpList 的大小剛好為一，且裡面的gate很多，由於我的

fraig在設計上是要處理完一個 fecGrp ，再去決定要不要resimulate，因此在這個狀況，就很難利用resimulate，去處理當前的Grp，而要一對一對pari慢慢看。

另一方面，由於有放棄整個grp的機制，在這樣的case下，很有可能做到一半，就放棄整個Grp了。

e) 其他可能優化的方式

整個 group 一起證明

由於如果對於一個group中所有的gate都要證一遍，則需要C n取二次的證明，然而如果很有可能整個group 是一樣的，則我們可以將全部的 gate 兩兩先 Xor 起來，再將所有的 Xor，Or起來當做output，然後試證 output = true，如果SAT ，則代表group其中至少有一個人和大家不一樣，若UNSAT 則大家都一樣。

這樣的做法好處在於，假設今天一個group中 gate 大部分都一樣時，可以省很多時間，而因為UNSAT 拿到的counter-example 也能在下次遇到這個 fecGrp 時，把不一樣的gate刪掉，然而，當一個group中大家都不一樣時，這個做法會比原本一個一個證明還要花上更多的時間，但依然是 O(n^ 2)。