DSnP Final Project Report

電機三 林孟瑾 b04901009
b04901009@ntu.edu.tw
0936-141-249

一、資料結構

在整個 final project 中，每個 gate 的資訊都存在 CirGate 這個 class 裡面，而每個 gate 的指標都記錄在 gate_list 這個 vector 裡。其中，CirGate 包含了 gate 的 id, gate 的 input1 的 id 及其在 gate_list 中的位置 input1pos, gate 的 input2 的 id 及其在 gate_list 中的位置 input2pos。此外，CirGate 的也紀錄了 gate 的類型，如果 gate 的類型為 PO_GATE，其 input 只會有一個，而如果 gate 的類型為 PI_GATE，則為沒有 input。CirGate 比較特別的地方在於，記錄了 output 的 vector，以方便 fanout 時能夠快速連接外接的 gate。其中也紀錄了每個 gate 在 simulation 時的數值，如此一來進行 simulation 時，如果遇到某個 gate 的 simulation 值已經改變，表示此 gate 已被 simulation，即可跳過，以便節省時間。

在 CirMgr 這個 class 裡，除了紀錄整個電路的 gate_list 之外，還紀錄了 map <int,int>gate_id_map，紀錄每個 gate 的 id 及其對應到 gate_list 的位置，由於許多部分需要從 id 對應到 gate_list 位置的資訊，為了節省時間，決定用 map 來儲存。而 net_list，儲存的則是 DFS list 中，gate 在 gate_list 的位置。此外，CirMgr 裡的 vector<size_t>simnum 是 simulation 時，要 assign 給每個 PI 的數值，之所以定義在 CirMgr，如此一來在呼叫 simulation 的 funtion 時，不需 pass PI 的數值，只需從 gate 的 input 的 id 即可找到對應的 simnum 位置而取得需要 assign 的數值，以便節省時間。CigMgr 也紀錄了 vector<vector<int>>fecs，為 simulation 後的 FEC groups，以每個 gate 在 gate_list 中的 index 來紀錄。

許多資料在儲存時，不紀錄 CirGate 的指標而紀錄 gate 在 gate_list 中的位置的原因為，map 在紀錄指標有時會出現錯誤，為了避免此一情況發生，我在設計資料結構時，除了 gate_list 此一 vector 儲存 CirGate 的指標之外，其餘的資料都是儲存“gate 在 gate_list 的 index”。

二、演算法

（一）sweep

1. 首先，用 net_map 紀錄 net_list 中的 gate。如此以來，判斷某個 gate 是否在 net_list 裡面就不需要 O(n)的時間複雜度去搜尋。
2. 檢查整個 gate_list，如果某個 gate 不在 net_map 裡面，且該 gate 不為 PI、PO、CONST，從 gate_list 之中移除。將此 gate 紀錄在陣列裡。
3. 如果 gate_list 中的 gate 位於 net_map 裡面，clear 此 gate 的 output，並將此 gate 存在 remain_gate 裡。更新 gate_id_map 為新的 index（因為 gate_list 的資訊有變動）。運用 index1 和 index2 分別紀錄 remain_gate 的數量跟整體處理過的 gate 的數量。

```
int index1=0 , index2=0;
while(index1<gate_list.size())
{
    if(net_map.count(index1)==0 && gate_list[index1]->getGateType()!=PI_GATE
       && gate_list[index1]->getGateType()!=PO_GATE
       && gate_list[index1]->getGateType()!=CONST_GATE)
    {
        haha.push_back( gate_list[index1] );
        gate_list.erase(gate_list.begin()+index1);
    }
    else{
        gate_list[index1]->clearOutput();
        remain_map[index2] = index1;
        gate_id_map[gate_list[index1]->getId()] = index1;
        index1++;

    }
    index2++;
}
```

4. 因為 gate_list 已變動，所以需要重新整理 net_list 儲存的“gate 在 gate_list 裡的位置”。
5. 最後，整理所有 gate 的 ouput 以及 input。

```
for(int i=0;i<gate_list.size();++i)
{
    if(gate_list[i] -> getGateType() == AIG_GATE)
    {
        int input1 = gate_list[i]->getInput1Id();
        int input2 = gate_list[i]->getInput2Id();
        if(input_id.count(input1)==true){
            gate_list[input_id[input1]]->setOutput( i ,gate_list[i]->getInput1Invered());
            gate_list[i]->setInput1Pos(input_id[input1]);
        }
        else if(gate_id_map.count(input1)==true)
        {
            gate_list[gate_id_map[input1]]->setOutput( i ,gate_list[i]->getInput1Invered());
            gate_list[i]->setInput1Pos(gate_id_map[input1]);
        }

        if(input_id.count(input2)==true){
            gate_list[input_id[input2]]->setOutput( i ,gate_list[i]->getInput2Invered());
            gate_list[i]->setInput2Pos(input_id[input2]);
        }
        else if(gate_id_map.count(input2)==true)
        {
            gate_list[gate_id_map[input2]]->setOutput( i ,gate_list[i]->getInput2Invered());
            gate_list[i]->setInput2Pos(gate_id_map[input2]);
        }

    }
    else if(gate_list[i] -> getGateType() == PO_GATE)
    {
        int input1 = gate_list[i]->getInput1Id();
        if(input_id.count(input1)==true){
            gate_list[input_id[input1]]->setOutput( i ,gate_list[i]->getInput1Invered());
            gate_list[i]->setInput1Pos(input_id[input1]);
        }
        else if(gate_id_map.count(input1)==true)
        {
            gate_list[gate_id_map[input1]]->setOutput( i ,gate_list[i]->getInput1Invered());
            gate_list[i]->setInput1Pos(gate_id_map[input1]);
        }

    }

}
```

（二）optimize

從 DFS list 中的 PO gate，運用 recursive 的方式來進行 optimize。

```
void
CirMgr::recursive_opt(int index)
```

1. 如果 gate 是 PO gate，呼叫 recursive_opt(input1 在 gate_list 的 index)
2. 如果是 AIG gate，先呼叫 recursive_opt(input1 在 gate_list 的 index)和 recursive_opt(input2 在 gate_list 的 index)，然後再進行化簡：

分別判斷兩個 input 是否為可化簡的情形，如果可以化簡，將此 gate 的 output 的 input 重接，也就是重新連接“含有此 gate 為 input 的 gate”。假設此 gate 化簡後需要刪除，將此 gate 的類型改成 TMP_GATE。而如果需要改成 const gate，將此 gate 類型改成 CONST_GATE。

3. 完成 recursive 的 optimize 之後，檢查整個 gate_list 裡，如果 gate 類型為 TMP_GATE 或是“類型為 CONST_GATE 且其位置非原本 const gate 的位置（表示此 gate 為多餘的 const gate）”，即可刪除此 gate。反之，clear gate 的 output 並且重新整理 gate_id_map。Index1 和 index2 的設計跟 sweep 時的意思相同。

```
int const_gate_ii = gate_id_map[const_gate_index];
int index1=0 , index2=0;
gate_id_map.clear();
while(index1<gate_list.size())
{
    if(gate_list[index1]->getGateType()==TMP_GATE || (index2!=const_gate_ii && gate_list[index1]->getGateType()==CONST_GATE ))
    {
        gate_list.erase(gate_list.begin()+index1);
        gate_number--;
    }
    else{
        gate_list[index1]->clearOutput();
        gate_id_map[gate_list[index1]->getId()] = index1;
        index1++;

    }
    index2++;
}
```

4. 最後，重新接 net_list 和所有 gate 的 input 以及 output。

（三）strash

遇到的問題：

處理 input 的反向資訊需要多存一個 bool，為了節省記憶體，改成將 input 的 gate 指標 cast 成 size_t 之後再做處理。

整體的大架構為一個 while loop，以 has_remove 此一 bool 來判斷 while loop 是否需要繼續執行，一開始的 has_remove 設成 true，每次 while 判斷如果 has_remove 為 true 就繼續執行，false 的話就停止。每一次的 while loop 檢查整個 net_list：

檢查整個 net_list：（檢查 net_list 前，has_remove 先設為 false）

1. 檢查整個 net_list 中的 gate，將 a 和 b 分別設成 gate 的 input1 和 input2，如果一個 gate 的“input1 在 gate_list 中的 index”存在於 remove_gate 裡，則將 a 設成 remove_gate[input1 在 gate_list 中的 index]，也就是設成其之後被取代成的 gate（remove_gate 為一個 map，存的是之後要刪除的 gate 和取代“被刪除的 gate”的 gate）。如果一個 gate 的“input2 在 gate_list 中的 index”存在於 remove_gate 裡，則將 b 設成 remove_gate[input2 在 gate_list 中的 index]，也就是設成其對應到的 gate。如此一來才能統一找到 equivalent 的 gate，不然如果某組 gate 雖然“意思相同”，但沒有機制判斷他們是否真正相同，會被誤認為不同的 gate。

2. 接著判斷這組 a 和 b 是否在 gate_hash 裡，map<pair<size_t, size_t>, int>gate_hash 存的是唯一一組 inputs 以及對應到的“(如果某個 gate 的 inputs 對應到此組 inputs pair)所需被取代的 gate”。如果此組 pair 在 gate_hash 裡，表示此 gate 有和其他 gate 相同的結構性，再繼續判斷此 gate 是否在 remove_gate 裡，如果在，表示他已經被標記為之後要 remove，如果不在，則新增一個 remove[gate 的 index]=gate_hash[inputs 的 pair]，表示之後這個 gate 要被取代成 gate_hash[inputs 的 pair]。然後把 has_remove 設成 true，表示 while 要繼續執行，繼續尋找要被 remove 的 gate。

3. 如果 gate_hash 裡沒有這組 inputs pair，則新增 gate_hash_[inputs pair]=gate 的 index，作為之後判斷 gate 是否與其他 gate 有相同結構性的依據。

```
if(gate_hash.count(make_pair(p1,p2))==true )
{
    if(gate_hash[make_pair(p1,p2)]!=net_list[i]){
        if(remove_gate.count(net_list[i])==false)
        {

            remove_gate[ net_list[i] ] = gate_hash[make_pair(p1,p2)];
            has_remove = true;
        }
    }
}
else if(gate_hash.count(make_pair(p2,p1))==true )
{
    if(gate_hash[make_pair(p2,p1)]!=net_list[i]){
        if(remove_gate.count(net_list[i])==false)
        {

            remove_gate[ net_list[i] ] = gate_hash[make_pair(p2,p1)];
            has_remove = true;
        }
    }
}
else{
    gate_hash[make_pair(p1,p2)]=net_list[i];
}
```

4. 判斷完全部 net_list 後，如果 has_remove 仍為 false，表示每個需要被刪除的 gate 都已被找到，while 即可停止。

結束 while 的檢查之後，進入 remove gate 以及重接階段：
1. 檢查整個 net_list，如果其中某個 gate 在 remove_gate 裡，表示之後要被刪除，改變其類型為 TMP_GATE，作為標示。
2. 檢查整個 net_list，如果其中某個 gate 的 input 出現在 remove_gate 裡，將此 input 改成 remove_gate 所對應到的 gate，也就是取代成另一個 gate（因為原本的 input 接到的 gate 之後會被刪除）。
3. 刪除 TMP_GATE，其餘留著的 gate 清除其 output 並整理 gate_id_map。
4. 最後，重新接 net_list 和所有 gate 的 input 以及 output。

整個 strash 的過程中，remove_gate 的 map 紀錄的是之後要被刪除的 gate 以及取代該被刪除的 gate 的 gate，而 gate_hash 存的是一組 inputs pair 和其對應到的 gate。由於有可能有超過兩個 gate 都是結構性相同而可以取代的 gate（意思就是，多個即將被刪的 gate 對應到同一個 gate，假設為 A），所以判斷一個 gate 的 input 是否和 gate_hash 存的 gate 的 input 相同時，我先判斷該 gate 的 input 是否存在在 remove_gate 中，如此一來，便可得知此 input 結構性相同的 gate 為 A gate，再以這個新的 inputs pair 作為判斷的依據。

另外，由於 input 還包括了 phase 的資訊，invert 與否需包含在 inputs pair 裡，所以我將 input 的 gate 的指標 cast 成 size_t 之後，如果有反向則加一，沒反向則維持原值，再將 input1 和 input2 得到的 size_t 組成 pair 然後存在 gate_hash 裡，如此便能確保 input pair 在 gate_hash 裡的唯一性。

```
if(gate_list[net_list[i]->getInput1Invered()==true){
    p1 = (size_t)((size_t)(     gate_list[  gate_list[net_list[i]->getInput1Pos()  ]   ))+1;
}
else{
    p1 = (size_t)((size_t)(     gate_list[  gate_list[net_list[i]->getInput1Pos()  ]   ));
}
```

（四）simulation

遇到的問題：

1. simulation 在反向時有時會出錯

->原本是使用”~”來做 inverse，後來查到 bitset function，使用 bitset 做反向問題即解決。

2. simulation 過久，無法有效處理何時停止 simulation

->使用 no_new_grp 以及 about_to_stop “動態控制”停止點，如果某次 simulation 剛好沒有分出新的 FEC group ，並不一定代表確實分完，所以用 about_to_stop 紀錄連續沒分出新的 group 的次數，如果超過 stop_range，過程即停止，其中如果又分出了新的 group，about_to_stop 的數值需要重新計算，如此一來，便能提升分出 FEC group 的效率。

random simulation

1. 首先，以 bool no_new_grp 紀錄 FEC group 是否生成，來決定 simulation 是否繼續執行。如果 no_new_grp 為 false，繼續執行。反之，以 about_to_stop 紀錄“no_new_grp 為 true”持續了幾次，如果持續的次數超過 stop_range 的話，simulation 就停止。Stop_range 的定義是以 gate_list 的大小決定。
2. 每次 simulation 時，先隨機產生 64 個長度為 input number 的 string，再從中製作 input number 個 64 bit 的 simulation pattern。如果 input number 為 100，第一個 PI 的 simulation pattern 就是“這 64 個 string 的第一位”所組成的 64 bit 值，第 100 個 PI 的 simulation pattern 就是“這 64 個 string 的第 100 位”所組成的 64 bit 值。

```
for(int i=0;i<input_number;++i)///input num
{
    size_t tmp = 0;
    int index = 0;
    if(sim_file.size()<64)
    {

        for(auto&j:sim_file)
        {
            size_t ia = (size_t)(j[i] - '0');
            tmp ^= ((size_t)(ia << index));
            index++;
        }
        for(int j=sim_file.size();j<64;++j)
        {
            tmp ^= ((size_t)(0<<j));
        }
    }
    else{

        for(auto&j:sim_file)
        {
            if(index>=64){break;}
            size_t ia = (size_t)(j[i] - '0');
            tmp ^= ((size_t)(ia << index));
            index++;
        }
    }
    simnum.push_back(tmp);
}
```

3. 接著，從 PO 開始，使用 recursive 的作法進行 simulation，

```
size_t
CirMgr::simulation(size_t index,bool inv)
```

index 代表的是 gate 在 gate_list 的 index，而 inv 則代表是否反向。如果某個 AIG gate 的 simulation value 已被改變（default 設為-123），則 return 此值（如果反向就 invert 此值）。反之，則呼叫”simulation(input1 的 index, input1 是否反向)& simulation(input2 的 index, inpu21 是否反向)”並設定此 gate 的 simulation 值為此值，然後 return 此值。而 PO gate 的話則只需判斷 input1。PI gate 的話，如果其 simulation value 有值就直接 return，反之，則設定其 simulation value 為對應的 simnum 的值，也就是該 input 的 simulation pattern。

```
size_t tmp_value=0;
    if(cirMgr->gate_list[index]->getGateType() == PI_GATE)
    {
        if(cirMgr->gate_list[index]->getSimValue()!=((size_t)(-123))){

            if(inv==true){
                return (size_t)(invert_value( cirMgr->gate_list[index]->getSimValue() ));
            }
            else{
                return  (size_t)(cirMgr->gate_list[index]->getSimValue());
            }
        }
        else if(inv==true){
            size_t aa=  ((simnum[ input_id[   cirMgr->gate_list[index]->getId()  ] ]));

            cirMgr->gate_list[index]->setSimValue( aa  );
            return  (size_t)(invert_value(   simnum[ input_id[   cirMgr->gate_list[index]->getId()  ] ]  ));
        }
        else {
            size_t aa=  ((simnum[ input_id[   cirMgr->gate_list[index]->getId()  ] ]));

            cirMgr->gate_list[index]->setSimValue( aa );
            return  (size_t) ( (simnum[ input_id[   cirMgr->gate_list[index]->getId()  ] ]) );
        }
    }


    else if(cirMgr->gate_list[index]->getGateType() == AIG_GATE){
        if(cirMgr->gate_list[index]->getSimValue()!= ((size_t)(-123)) ){

            if(inv==true){
                return (size_t)(invert_value( cirMgr->gate_list[index]->getSimValue() ));
            }
            else{
                return   (size_t)((cirMgr->gate_list[index]->getSimValue()));
            }
        }
        else if(inv==true){

            tmp_value = (  (simulation( cirMgr->gate_list[index]->getInput1Pos(),cirMgr->gate_list[index]
                ->getInput1Invered()) & simulation( cirMgr->gate_list[index]->getInput2Pos(),cirMgr->
                gate_list[index]->getInput2Invered())  ) );

            cirMgr->gate_list[index]->setSimValue(  tmp_value  );
            return (size_t)(invert_value(tmp_value));

        }
        else{

            tmp_value =  (size_t)(simulation( cirMgr->gate_list[index]->getInput1Pos(),cirMgr->gate_list
                [index]->getInput1Invered()) & simulation( cirMgr->gate_list[index]->getInput2Pos(),
                cirMgr->gate_list[index]->getInput2Invered()));
            cirMgr->gate_list[index]->setSimValue(tmp_value );
            return  tmp_value;

        }
    }
```

4. simulation 完之後，每個 gate 都會有 simulation value，此時，如果 fecs 為空的（fecs 為 vector<vector<int>>，用來紀錄多個 FEC group，其中，int 為 gate 在 gate_list 的 index），表示 FEC group 尚未開始紀錄，先使用 map<size_t,vector<int>>sim_result，判斷哪些 gate 的 simulation value 相同。其中，如果 gate 的 simulation 的 inverse 相同也需紀錄，設計上將其 index 減掉 gate_list.size()，變成負數，以方便之後的判斷。然後再將 sim_result 中的許多 FEC group 放進 fecs 裡。

5. 進行下一次 simulation 時，由於 fecs 裡已經有 FEC groups，只要判斷 group 裡的 gate，如果新的 simulation value 不同，就分出新的 FEC group。實作上使用 hash_map<size_t,vector<int>>tmp_group，判斷一個 group 裡的 gate 是否出現不同的 simulation value。而 vector<vector<int>>tmp_list 則是紀錄新的 FEC groups，如果沒有新的 group 分出來，則 fecs 不需要改變，反之，將 fecs 設為 tmp_list。

```
no_new_grp=true;
vector<vector<int>>tmp_list;
for(auto&i:fecs)
{

    hash_map<size_t,vector<int>>tmp_group;
    tmp_group.clear();
    for(auto&j:i)
    {
        if( j<0){
            size_t ss = invert_value(  gate_list[j+gate_list.size()]->getSimValue()  );
            tmp_group[ss].push_back(j);
        }
        else{
            if(tmp_group.count(gate_list[j]->getSimValue())==false)
            {
                tmp_group[gate_list[j]->getSimValue()].push_back(j);
            }
            else{
                tmp_group[gate_list[j]->getSimValue()].push_back(j);
            }
        }
    }

    if(tmp_group.size()>1)
    {
        no_new_grp = false;
        for(auto&j:tmp_group)
        {
            if(j.second.size()>1)
            {
                vector<int>ss;
                for(auto&k:j.second)
                {
                    ss.push_back(k);
                }
                tmp_list.push_back(ss);
            }

        }
    }
    else{
        tmp_list.push_back (i);
    }
}

if(no_new_grp == false){fecs = tmp_list;}
```

5. 最後，將 fecs 排序，並把 FEC groups 中，含有 const gate 的 group 移到 fecs 的第一個位置。

6. 印出 FEC groups 時，如果 group 中的第一個 gate 的 index 為負值，表示當初紀錄的 simulation value 為反向，所以後面的 gate 在印出時，如果為反向則不需要印出”！”，如果為正向則需要印出”！”。

```
for(auto&i:fecs)
{


    cout<<"["<<ii<<"]";

    if(i[0]<0){
        for(auto&j:i)
        {

            if(j<0){  cout<<" "<<gate_list[j+gate_list.size()]->getId();  }
            else {cout<<" !"<<gate_list[j]->getId();}

        }

    }
    else{
        for(auto&j:i)
        {

            if(j<0){  cout<<" !"<<gate_list[j+gate_list.size()]->getId();  }
            else {cout<<" "<<gate_list[j]->getId();}

        }

    }
    cout<<endl;
    ii++;
}
```

file simulation

file simulation 只有 simulation pattern 的生成和 random simulation 的方法有些許不同，其他有關 simulation 和 FEC groups 的處理的作法皆相同。

1. 將檔案讀進來並把每個 string 存起來為 vector<string>sim_file。
2. 接著進入 while 迴圈，每次從這些 string 中取 64 個 string，從中製作出 input number 個 64 bit 的 simulation pattern。如果不足 64 個，其餘 bit 即補上零。假設 input number 為 100，第一個 PI 的 simulation pattern 就是“這 64 個 string 的第一位”所組成的 64 bit 值，第 100 個 PI 的 simulation pattern 就是“這 64 個 string 的第 100 位”所組成的 64 bit 值。然後把這 64 個 string 從 sim_file 中刪除，如果少於 64 個 string，則全部 clear 掉。While 迴圈裡面接著進行 simulation 以及 FEC group 的處理，如果 sim_file 為空的時，while 即停止。
3. 處理 fecs 的排序以及含有 const gate 的 group。

（五）fraig

遇到的問題：

1. 將化簡的電路與 ref 化簡的電路 miter 之後的結果並不是 const

->發現是因為 cirwrite 時並未處理好 PO 的 function，處理完後即解決

1. 使用 SAT solver 證明 FEC groups 中兩兩 gate 是否 functionally equivalent。（全部的 pair 我都有證明一次）
2. 如果相同，則新增至 remove_gate 中（remove_gate 為 map，存的是“之後要被刪除的 gate”以及”取代被刪除的 gate 的 gate”）。並將之後要被刪除的 gate 的類型 assign 成 TMP_GATE。
3. 檢查整個 net_list，如果某個 gate 的 input 存在於 remove_gate 中，表示此 gate 要被取代成 remove_gate[input gate 的 index]。特別要注意的是，如果此 gate 的 input 在 remove_gate 中對應到的是”負的 index”，表示該 input 要被取代為 remove_gate[input gate 的 index]的”反向”，所以此 input 要多考慮 phase 轉變的影響。同樣的，如果 input 的 index 減掉 gate_list.size()之後的值存在於 remove_gate 中，表示該 input 要取代為其所對應到的 gate 的反向。

```
if(remove_gate.count( gate_list[net_list[i]]-> getInput1Pos())==true)
{
    int ss  = remove_gate[gate_list[net_list[i]]-> getInput1Pos()];
    if(ss<0)
    {
        gate_list[net_list[i]]-> setInput1Pos( ss+gate_list.size()  );
        gate_list[net_list[i]]-> setInput1(gate_list[  ss+gate_list.size()  ]->getId());
        gate_list[net_list[i]]-> setInput1Invered( (gate_list[net_list[i]]-> getInput1Invered())^true    );
    }
    else{
        gate_list[net_list[i]]-> setInput1Pos( ss);
        gate_list[net_list[i]]-> setInput1(gate_list[  ss  ]->getId());
    }

}
else if(remove_gate.count( (gate_list[net_list[i]]-> getInput1Pos())-gate_list.size())==true)
{
    int ss  = remove_gate[gate_list[net_list[i]]-> getInput1Pos()-gate_list.size()];

    if(ss<0)
    {
        gate_list[net_list[i]]-> setInput1Pos( ss+gate_list.size()  );
        gate_list[net_list[i]]-> setInput1(gate_list[  ss+gate_list.size()  ]->getId());

    }
    else{
        gate_list[net_list[i]]-> setInput1Pos( ss);
        gate_list[net_list[i]]-> setInput1(gate_list[  ss  ]->getId());
        gate_list[net_list[i]]-> setInput1Invered( (gate_list[net_list[i]]-> getInput1Invered())^true    );
    }

}
```

4. 檢查整個 gate_list，如果類型為 TMP_GATE 則刪除，反之，clear 該 gate 的 output，並重整 gate_id_map。重新整理 net_list 以及每個 gate 的 input 和 output 。

三、實際測試

1. sweep and optimize

```
sabrinadeMacBook-Pro:tests.fraig sabrina$ ../fraig
fraig> cirr opt07.aag

fraig> cirsw
Sweeping: AIG(5) removed...
Sweeping: UNDEF(6) removed...
Sweeping: AIG(7) removed...
Sweeping: AIG(8) removed...
Sweeping: AIG(9) removed...
Sweeping: AIG(10) removed...

fraig> usage
Period time used : 0 seconds
Total time used  : 0 seconds
Total memory used: 0.2695 M Bytes

fraig>
```

```
sabrinadeMacBook-Pro:tests.fraig sabrina$ ../ref/fraig-mac
fraig> cirr opt07.aag

fraig> cirsw
Sweeping: AIG(5) removed...
Sweeping: UNDEF(6) removed...
Sweeping: AIG(7) removed...
Sweeping: AIG(8) removed...
Sweeping: AIG(9) removed...
Sweeping: AIG(10) removed...

fraig> usage
Period time used : 0 seconds
Total time used  : 0 seconds
Total memory used: 0.2656 M Bytes

fraig>
```

```
sabrinadeMacBook-Pro:tests.fraig sabrina$ ../fraig
fraig> cirr opt01.aag

fraig> ciropt
Simplifying: 1 merging 2...

fraig> usage
Period time used : 0 seconds
Total time used  : 0 seconds
Total memory used: 0.25 M Bytes

fraig> cirr opt02.aag -r
Note: original circuit is replaced...

fraig> ciropt
Simplifying: 0 merging 2...

fraig> usage
Period time used : 0 seconds
Total time used  : 0 seconds
Total memory used: 0.2812 M Bytes

fraig> cirr opt03.aag -r
Note: original circuit is replaced...

fraig> ciropt
Simplifying: 1 merging 2...

fraig> usage
Period time used : 0 seconds
Total time used  : 0 seconds
Total memory used: 0.293 M Bytes

fraig> cirr opt04.aag -r
Note: original circuit is replaced...

fraig> ciropt
Simplifying: 0 merging 2...

fraig> usage
Period time used : 0 seconds
Total time used  : 0 seconds
Total memory used: 0.3047 M Bytes
```

```
sabrinadeMacBook-Pro:tests.fraig sabrina$ ../ref/fraig-mac
fraig> cirr opt01.aag

fraig> ciropt
Simplifying: 1 merging 2...

fraig> usage
Period time used : 0 seconds
Total time used  : 0 seconds
Total memory used: 0.2539 M Bytes

fraig> cirr opt02.aag -r
Note: original circuit is replaced...

fraig> ciropt
Simplifying: 0 merging 2...

fraig> usage
Period time used : 0 seconds
Total time used  : 0 seconds
Total memory used: 0.3125 M Bytes

fraig> cirr opt03.aag -r
Note: original circuit is replaced...

fraig> ciropt
Simplifying: 1 merging 2...

fraig> usage
Period time used : 0 seconds
Total time used  : 0 seconds
Total memory used: 0.3125 M Bytes

fraig> cirr opt04.aag -r
Note: original circuit is replaced...

fraig> ciropt
Simplifying: 0 merging 2...

fraig> usage
Period time used : 0 seconds
Total time used  : 0 seconds
Total memory used: 0.3203 M Bytes
```

```
sabrinadeMacBook-Pro:tests.fraig sabrina$ ../fraig
fraig> cirr opt05.aag

fraig> ciropt
Simplifying: 1 merging !6...
Simplifying: 0 merging 8...
Simplifying: 0 merging 9...
Simplifying: 7 merging !11...
Simplifying: 4 merging 10...

fraig> usage
Period time used : 0 seconds
Total time used  : 0 seconds
Total memory used: 0.2695 M Bytes

fraig> cirr opt06.aag -r
Note: original circuit is replaced...

fraig> ciropt
Simplifying: 0 merging 3...
Simplifying: 0 merging 4...
Simplifying: 0 merging 5...
Simplifying: 7 merging !8...
Simplifying: 7 merging !6...
Simplifying: 7 merging !11...
Simplifying: 0 merging 10...
Simplifying: 7 merging 9...
Simplifying: 0 merging 12...
Simplifying: 0 merging 13...
Simplifying: 7 merging !14...

fraig> usage
Period time used : 0 seconds
Total time used  : 0 seconds
Total memory used: 0.293 M Bytes
```

```
sabrinadeMacBook-Pro:tests.fraig sabrina$ ../ref/fraig-mac
fraig> cirr opt05.aag

fraig> ciropt
Simplifying: 1 merging !6...
Simplifying: 0 merging 8...
Simplifying: 0 merging 9...
Simplifying: 7 merging !11...
Simplifying: 4 merging 10...

fraig> usage
Period time used : 0 seconds
Total time used  : 0 seconds
Total memory used: 0.2812 M Bytes

fraig> cirr opt06.aag -r
Note: original circuit is replaced...

fraig> ciropt
Simplifying: 0 merging 3...
Simplifying: 0 merging 4...
Simplifying: 0 merging 5...
Simplifying: 7 merging !8...
Simplifying: 7 merging !6...
Simplifying: 7 merging !11...
Simplifying: 0 merging 10...
Simplifying: 7 merging 9...
Simplifying: 0 merging 12...
Simplifying: 0 merging 13...
Simplifying: 7 merging !14...

fraig> usage
Period time used : 0 seconds
Total time used  : 0 seconds
Total memory used: 0.3086 M Bytes
```

測試的結果發現，進行 sweep 和 optimize 所花的時間和記憶體皆與 reference 相差不多，且化簡結果皆和 ref 相同。

2. strash

測試 gate 數量較大的程式:

指令為 cirr sim10.aag，然後 cirstr

左邊為我的程式右邊為 reference

```
fraig> usage
Period time used : 0 seconds
Total time used  : 0 seconds
Total memory used: 0.5859 M Bytes
```

```
fraig> usage
Period time used : 0 seconds
Total time used  : 0 seconds
Total memory used: 0.4492 M Bytes
```

指令為 cirr sim09.aag，然後 cirstr

左邊為我的程式右邊為 reference

```
fraig> usage
Period time used : 0.01 seconds
Total time used  : 0.01 seconds
Total memory used: 1.586 M Bytes
```

```
fraig> usage
Period time used : 0 seconds
Total time used  : 0 seconds
Total memory used: 0.9648 M Bytes
```

時間方面沒有相差太多，而記憶體方面稍微多出 ref 一些。可能原因為使用 hash_map 存取許多資料造成記憶體的增加。

3. simulation and fraig

以 gate 數量最多的 sim13.aag 測試

我的程式 cirsim -r 測試了 114240 patterns，而 reference 測試了 94336 個。

我的分出了 3210 個 FEC groups 而 ref 分出 3526 個 FEC groups。

雖然我 simulation 的 pattern 較多，但可能因為其中的 pattern 有些會重複，影響分出 FEC group 的效果 。

左邊為我的程式右邊為 reference

```
fraig> usage
Period time used : 23.21 seconds
Total time used  : 23.21 seconds
Total memory used: 28.73 M Bytes
```

```
fraig> usage
Period time used : 9.47 seconds
Total time used  : 9.47 seconds
Total memory used: 18.11 M Bytes
```

經過 fraig 之後:

```
fraig> usage
Period time used : 133.9 seconds
Total time used  : 157.1 seconds
Total memory used: 55.47 M Bytes
```

```
fraig> usage
Period time used : 105 seconds
Total time used  : 114.5 seconds
Total memory used: 43.23 M Bytes
```

花了不少時間證明 SAT pairs，記憶體也隨之增加，因為使用了 remove_gate 來存取需要刪除的 gate。

Sim13.aag 進行 file simulation 結果如下：

```
fraig> usage
Period time used : 4.51 seconds
Total time used  : 4.51 seconds
Total memory used: 106.7 M Bytes
```

```
fraig> usage
Period time used : 2.63 seconds
Total time used  : 2.63 seconds
Total memory used: 18.86 M Bytes
```

記憶體用量非常多，估計是跟讀檔的存取有關。

接著進行 fraig:

```
fraig> usage
Period time used : 232.9 seconds
Total time used  : 232.9 seconds
Total memory used: 125.6 M Bytes
```

```
fraig> usage
Period time used : 135.9 seconds
Total time used  : 135.9 seconds
Total memory used: 44.89 M Bytes
```

左邊我的程式明顯地比 ref 花更多時間，可能的原因為我每個 FEC 的 pair 都會證明。而記憶體的部分，可能原因是因為讀檔時存的 sim_file 很大的關係，以及所有 hash_map 與 map 的使用都有可能因為 gate 數量的增加而花更多的記憶體。

化簡後的 cirp -n 顯示出我的剩下 84674 個 gate，而 ref 剩下 84023 個 gate，化簡的效果並未相差很多。

測試 gate 數量較小的程式

sim15.aag 總 gate 數量為 928 個。

我的程式 cirsim -r 測試了 4864 patterns，而 reference 測試了 5760 個。

我的分出了 30 個 FEC groups 而 ref 分出 30 個 FEC groups。且分出的 FEC groups 相同，表示在 gate 數量小的情況下，simulation 的效果不錯。

左邊為我的程式右邊為 reference

```
fraig> usage
Period time used : 0.01 seconds
Total time used  : 0.01 seconds
Total memory used: 0.5898 M Bytes
```

```
fraig> usage
Period time used : 0 seconds
Total time used  : 0 seconds
Total memory used: 0.5234 M Bytes
```

接著進行 fraig:

```
fraig> usage
Period time used : 0.02 seconds
Total time used  : 0.03 seconds
Total memory used: 0.9414 M Bytes
```

```
fraig> usage
Period time used : 0.02 seconds
Total time used  : 0.02 seconds
Total memory used: 0.9453 M Bytes
```

時間和記憶體的使用狀況和 ref 相差不多。

4.綜合測試

指令如下：

```
cirr sim09.aag
ciropt
cirp -n
cirstr
cirp -n
cirsim -f pattern.09
cirfraig
cirp -n
```

usage

測試結果如下：

```
fraig> usage
Period time used : 0.07 seconds
Total time used  : 0.07 seconds
Total memory used: 2.66 M Bytes
```

```
fraig> usage
Period time used : 0.06 seconds
Total time used  : 0.06 seconds
Total memory used: 1.676 M Bytes
```

其中，ciropt 和 cirstr 都有化簡掉許多 gate，且化簡後的 cirp -n 皆相同。

從 usage 中可以看出，時間相差不大且記憶體使用量只有多一些，效果尚可。

四、綜合討論

這些功能中，花最多時間設計以及 debug 的 function 為 simulation，需要細心以及清晰的邏輯才能完成，且 simulation 關係到 fraig 的效率，因此我投入了許多精力設計。寫完這份作業，不僅對程式設計更有了解，也體會到軟體工程師的生活，雖然每天花不少時間寫程式，但學到的東西真的很紮實。這學期的資結磨練了我的程式能力，練習了許多有趣的程式，也和同學一起討論切磋想法，我相信這些都是自學無法得到的，真的是一個十分寶貴的經驗！