Data Structure and Programming

資料結構與程式設計

授課老師：黃鐘揚

Final Project Report

Functionally Reduced And-Inverter Graph (FRAIG) Implementation by C++

NTUEE
B03901056
孫凡耕
0988029102

2016/1/21

➢ CirCuit Structure

Design of Class CirMgr：

Important data members inside CirMgr is shown as below：

CirMgr typedef myList < myList<unsigned>* > mmu

- ofstream* _simLog
- unsigned para[5],b_size
- vector<CirGate*> gate_list
- CirGate** line_list
- mmu* simt

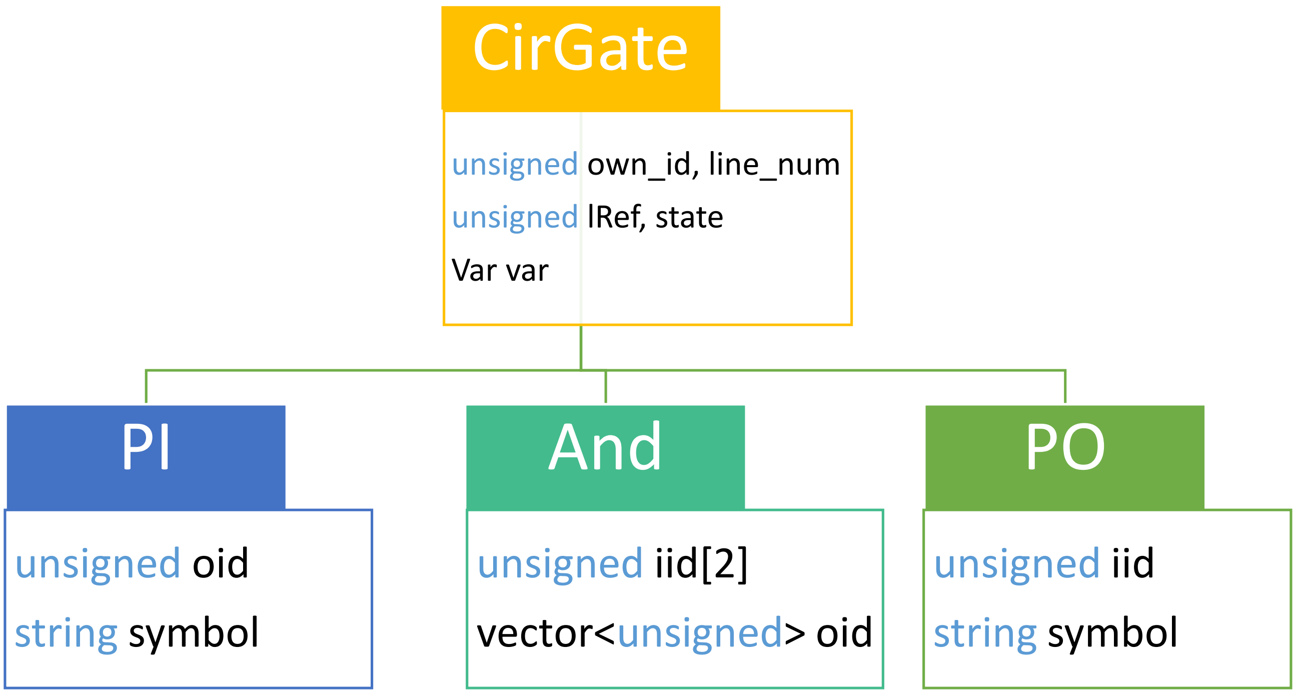
- _simLog → for output log file
- para[5] → corresponds to the five parameters in the aag file header
- b_size → the number of buckets of simt
- gate_list → stores all the gates according the gate id.
- line_list → stores all the gates according to the numbers of line declared in aag file.
- simt → the table during simulation

Note：

- **The method to store gates in gate_list：**
  If the id of a gate is 9, then it will be stored in gate_list[9]. This means that I will use vector.resize() to allocate the memory space first, then use = to assign the corresponding slot to the CirGate*. The numbers of slots I need is then para[0](M)+para[3](O). Thus, there will be posiibly 0(null) elements in gate_list. This promises constant time performance while parsing the circuit and finding the CirGate* according to the id.
- **The usage of line_list：**
  Also it is possibly to use only gate_list to operator some tasks about the line number, it's basically time-consuming. So I choose to sacrifice some memory space and store the data in this line_list. So the CirGate* declared in line 9 is stored in line_list[9]. For example, while determineing the symbolic name, i0 corresponds to the first PI "declared" in the aag file, not the PI with the smallest id. Without line_list, the fastest way to parse the symbolic name is sort the gate_list according the line_num I stored inside a gate, then assign the symbolic name. This is absolutely too slow (at least logarithmic time) compared to constant time. Other examples including printing PI, PO and writeaag, which all need to be sorted to match the results of the ref program if without line_list.
  By the way, since lines in aag file is continuous, there is no 0(null) elements in line_list(except line 0, line 1). So I use CirGate**.

Design of Class CirGate：

I implemented CirGate as the base class of class PI, PO and And.
The data members inside each class are shown as below：



In CirGate：

- own_id → the id of the gate itself
- line_num → the number of line where the gate is defined in the aag file
- lRef → means local reference，for depth-first-search traversal
- state → record the results of this gate after simulation
- Var → for SatSolver

In PI：

- oid → the id of the output gate
- symbol → the symbolic name of PI

In And：

- iid[2] → two LITERAL ids of the input gates
- oid → output id(s) of the output gate (possibly >> 1, usually 1 or 2)

In PO：

- iid → the LITERAL id of the input gate
- symbol → the symbolic name of PO

Note：

- I store the LITERAL input id(s) in And and PO, this has many advantages：

1. I don't need extra bool inside a class to judge if its input is inverted, instead I use

((iid[k]/2)*2 != iid[k]). So easier to maintain.
2. While optimizing, easier to judge if input has const0/const1 or is same/opposite.
3. My HashKey also depends upon the LITERAL id, not gate id.

- I don’t implement class UNDEF, so there may be some minor difference between the outcome of my program and ref program (only when cout). However, there is some way to solve this problem, which I will mention later.

The advantages of this structure：

1. **Fast：**
   I only need to modify the iid and oid in the derived gates when rearranging the structure of the circuit. This basically guarantees a wonderful performance in almost every command, including parsing the circuit (parts included in HW#6) and simplifying the circuit(parts included in HW#7).
2. **Use less memory：**
   I don’t need to store pointer CirGate* pointing to the input/output of a gate alongside with iid and oid, thus using less memory. This also contributes to a faster performance.
3. **Easier to maintain：**
   Since iid and oid are the only factor that influence the parsing of the circuit, its easier to maintain while rewiring and merging the circuit. This also contributes to a faster performance.

The Disadvantages of this structure：

1. **No direct access to pointer(minor effects)：**
   Since I only stored id(s) in the class, when I have to access to the pointer and perform some tasks on OTHER gates, then I must write gate_list[id] in order to acquire the CirGate* pointer. Inside cirGate.cpp, I have to use the global variable CirMgr* cirMgr. However, fundamentally this only makes the codes a bit longer, but doesn’t really affect the performance. Also, this doesn’t makes coding more difficult. So I think this is only a minor disadvantages comparing to the huge advantages as mentioned above.

➢ Algorithm

Sweep：

1. Steps

i. Use depth-first-search traversal, starting from every PO to go through the circuit and set mark on visited gate.
ii. For all the elements in gate_list that is a valid pointer and is a And gate and is not marked, rewire the circuit that is neighboring to gate and update gate_list.
iii. If the elements is not a valid pointer and is not marked, simply cout the unsigned stored in gate_list (explained later).

2. *Pseudo-code*：

```
CirMgr::sweep()
{
    for( every PO )
        depth-first-search traversal(POs);    //set mark while traversal
    for( every gate_list[i] except PO )
    {
        if( gate_list[i] is a valid CirGate* pointer and is a And gate and is not marked)
        {
            cout << "Sweeping: AIG(" << gate_list[i]->own_id() << ") removed...\n";
            rewire_circuit(i);
            delete gate_list[i];
            gate_list[i] = 0;
        }
        else( gate_list[i] is a UNDEF and its output is not marked )
             cout << "Sweeping: UNDEF(" << i << ") removed...\n";
    }
}
```

Comments about UNDEF：

Since I have no class UNDEF gate implementation, the corresponding slot in the gate_list will be 0(NULL). Instead of building a class UNDEF, I choose to use type cast to store the "output" id of the UNDEF gate in the originally 0(null) slot in gate_list. For example, if a gate with id 7 has a UNDEF input with id 5 and another input with id 6 Then gate_list[7] and gate_list[6] will stored CirGate* pointer and gate_list[5] will be NULL originally. (gate_list[7]->input_id still stored 5 and 6). During parsing, I use type conversion → (size_t)gate_list[6] = 7, and stored the "output" id of the UNDEF gate in gate_list[6]. So now gate_list[6] is not 0(NULL), but its value is actually an unsigned int. I can do this because the value of a valid pointer is typically very large (>> 10^10 in my computer) and an gate id will never exceed the value.

Then, when I want to cout << "Sweeping :UNDEF….", I can simply write：

```
If(   gate_list[i] is not a valid CirGate* pointer and gate_list[(size_t)gate_list[i]] is not marked)
    cout << "Sweeping: UNDEF(" << i << ") removed...\n";
```

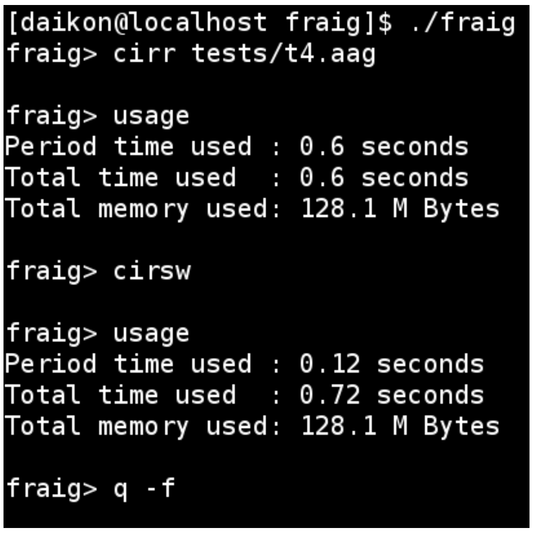
Note the meaning of gate_list[(size_t)gate_list[i]].

3. *Experiments*：*(aag file generated by AAGgen)*

i. **A huge circuit with nothing to sweep**：

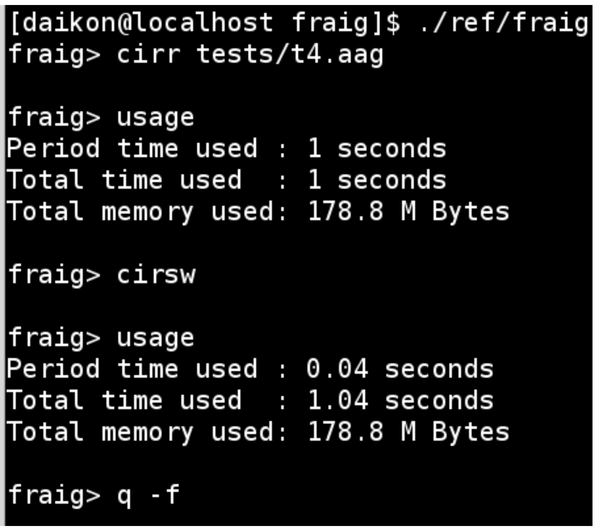
aag 999999 200000 0 338957 799999

my program：



```
[daikon@localhost fraig]$ ./fraig
fraig> cirr tests/t4.aag

fraig> usage
Period time used : 0.6 seconds
Total time used  : 0.6 seconds
Total memory used: 128.1 M Bytes

fraig> cirsw

fraig> usage
Period time used : 0.12 seconds
Total time used  : 0.72 seconds
Total memory used: 128.1 M Bytes

fraig> q -f
```

ref program：



```
[daikon@localhost fraig]$ ./ref/fraig
fraig> cirr tests/t4.aag

fraig> usage
Period time used : 1 seconds
Total time used  : 1 seconds
Total memory used: 178.8 M Bytes

fraig> cirsw

fraig> usage
Period time used : 0.04 seconds
Total time used  : 1.04 seconds
Total memory used: 178.8 M Bytes

fraig> q -f
```

Conclusion：

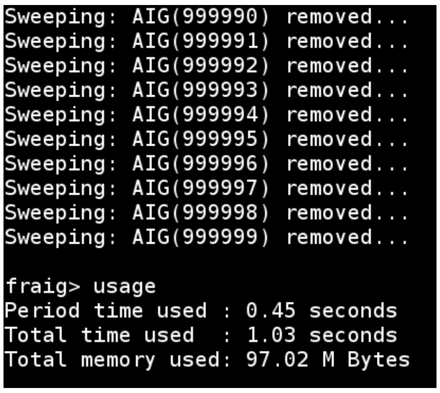
My program runs faster while parsing the circuit(actually in every case, my program always runs faster in parsing, so will not discuss in later cases), but the ref program go through every gate and make sure no gate can be swept in a faster manner.

My program also uses less memory (for every function and test cases, my program and ref program consumes about the same amount of memory space, and since memory consumption is not an issue for FRAIG implementation, we will no discuss about this in later cases).

ii. **A huge circuit with every And to be sweeped：(the only PO connected to const0)**

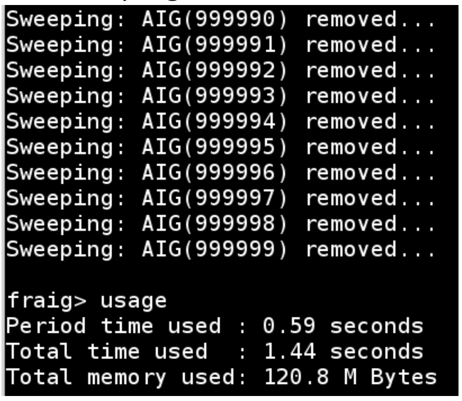
aag 999999 200000 0 1 799999

my program：



```
Sweeping: AIG(999990) removed...
Sweeping: AIG(999991) removed...
Sweeping: AIG(999992) removed...
Sweeping: AIG(999993) removed...
Sweeping: AIG(999994) removed...
Sweeping: AIG(999995) removed...
Sweeping: AIG(999996) removed...
Sweeping: AIG(999997) removed...
Sweeping: AIG(999998) removed...
Sweeping: AIG(999999) removed...

fraig> usage
Period time used : 0.45 seconds
Total time used  : 1.03 seconds
Total memory used: 97.02 M Bytes
```

ref program：



```
Sweeping: AIG(999990) removed...
Sweeping: AIG(999991) removed...
Sweeping: AIG(999992) removed...
Sweeping: AIG(999993) removed...
Sweeping: AIG(999994) removed...
Sweeping: AIG(999995) removed...
Sweeping: AIG(999996) removed...
Sweeping: AIG(999997) removed...
Sweeping: AIG(999998) removed...
Sweeping: AIG(999999) removed...

fraig> usage
Period time used : 0.59 seconds
Total time used  : 1.44 seconds
Total memory used: 120.8 M Bytes
```

Conclusion：

My program runs a little faster when there are enormous quantity of gates to be swept.

4. Performance：

My program and ref program basically runs at same speed, but there is still some difference. For a huge circuit(>10^6). If there is nothing to sweep, my program runs slower, but if there are many gates to sweep, then my program runs faster. This means that ref program has a better judgment (if/else) but my program excel at rewiring. By the way, my program always use less memory then the ref. The reason of the above results is due to my implementation of CirGate as mentioned.

Trivial Optimize：

1. *Steps*

i. While depth-first-search traversal-ing, visit all the input gates of a gate.
ii. After visiting all the input gates of a gate, judge if this particular gate needs to be optimize.
iii. If need to be optimize, rewire the circuit that is neighboring to gate and update gate_list.

2. *Pseudo-code*：

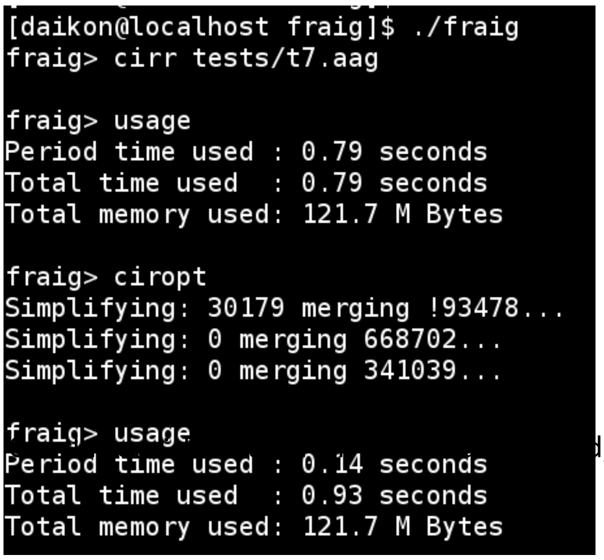
```
CirMgr::optimize()
{
    for( every PO )
        depth_first_search_traversal_for_optimize(POs); //set mark while traversal
}
CirMgr::depth_first_search_traversal_for_optimize( CirGate* this_gate )
{
    for( two input gates of this_gate )
    {
        if( the input gate is valid and not marked)
        {
            set mark on input gate;
            depth_first_search_traversal_for_optimize( this_gate → input gate);
        }
    }
    If( _move need to be optimize )
        rewire(this_gate);
}
```

3. *Experiments*：*(aag file generated by AAGgen)*

i. A huge circuit with few gates required optimization：

aag 999999 260000 0 344933 739999

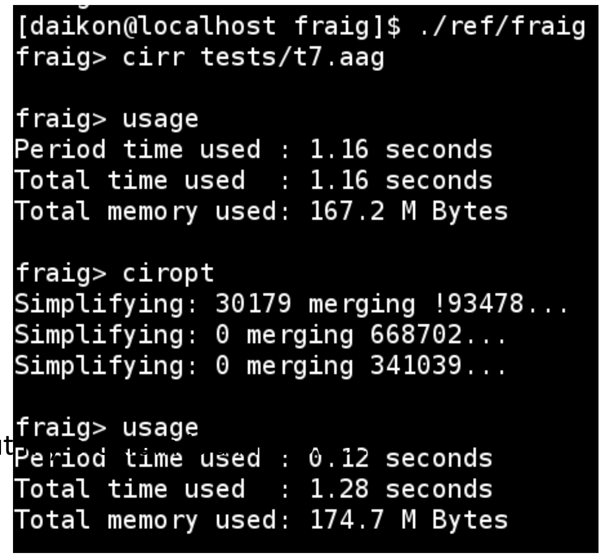
my program：



```
[daikon@localhost fraig]$ ./fraig
fraig> cirr tests/t7.aag

fraig> usage
Period time used : 0.79 seconds
Total time used  : 0.79 seconds
Total memory used: 121.7 M Bytes

fraig> ciropt
Simplifying: 30179 merging !93478...
Simplifying: 0 merging 668702...
Simplifying: 0 merging 341039...

fraig> usage
Period time used : 0.14 seconds
Total time used  : 0.93 seconds
Total memory used: 121.7 M Bytes
```

d, but

ref program：



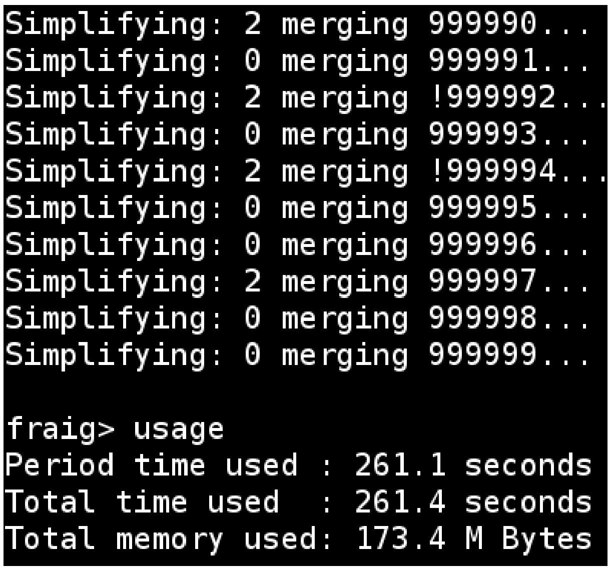
```
[daikon@localhost fraig]$ ./ref/fraig
fraig> cirr tests/t7.aag

fraig> usage
Period time used : 1.16 seconds
Total time used  : 1.16 seconds
Total memory used: 167.2 M Bytes

fraig> ciropt
Simplifying: 30179 merging !93478...
Simplifying: 0 merging 668702...
Simplifying: 0 merging 341039...

fraig> usage
Period time used : 0.12 seconds
Total time used  : 1.28 seconds
Total memory used: 174.7 M Bytes
```

ii. A huge circuit with most gates required optimization：

(change one line in AAGgen.cpp and produces a circuit with many same or opposite inputs)

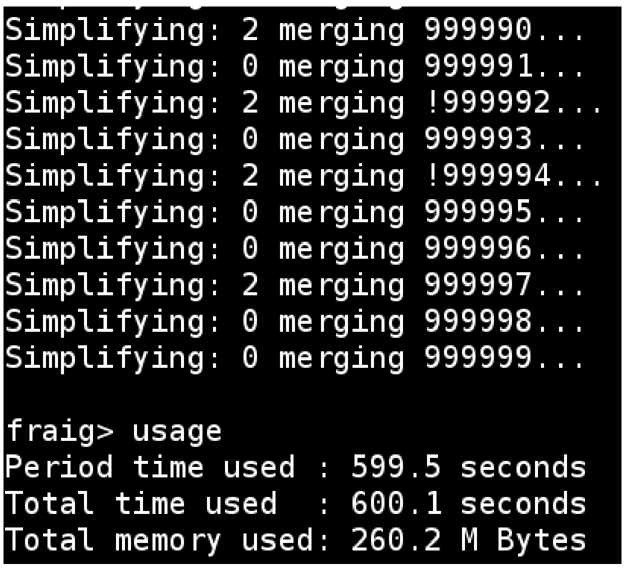
aag 999999 280000 0 999998 719999

my program：



```
Simplifying: 2 merging 999990...
Simplifying: 0 merging 999991...
Simplifying: 2 merging !999992...
Simplifying: 0 merging 999993...
Simplifying: 2 merging !999994...
Simplifying: 0 merging 999995...
Simplifying: 0 merging 999996...
Simplifying: 2 merging 999997...
Simplifying: 0 merging 999998...
Simplifying: 0 merging 999999...

fraig> usage
Period time used : 261.1 seconds
Total time used  : 261.4 seconds
Total memory used: 173.4 M Bytes
```

ref program：



```
Simplifying: 2 merging 999990...
Simplifying: 0 merging 999991...
Simplifying: 2 merging !999992...
Simplifying: 0 merging 999993...
Simplifying: 2 merging !999994...
Simplifying: 0 merging 999995...
Simplifying: 0 merging 999996...
Simplifying: 2 merging 999997...
Simplifying: 0 merging 999998...
Simplifying: 0 merging 999999...

fraig> usage
Period time used : 599.5 seconds
Total time used  : 600.1 seconds
Total memory used: 260.2 M Bytes
```

Conclusion：My program runs two times faster than ref program.

5. Performance：

My program and ref program basically runs at about the same speed, but the larger the circuit is, the greater the advantage of my program over ref program is. The speed of my program is approximately two times faster than ref program.

Strashing：

■ My implementation of Hash

1. Code: (I only show the part that I used in strashing)

```
Class HashKey
{
public：
    HashKey(unsigned i0, unsigned i1) : i0(i0), i1(i1) {}
    size_t operator() () const
    {
        if( ( (i0/2)*2 !=i0 ) ^ (i1/2)*2 != i1) ) return i0/2;
        else return i1/2;
    }
    bool operator == (const HashKey& k) const { return (k.i0==i0 && k.i1==i1); }
private:
    size_t i0, i1;
}
```

```
template <class HashKey, class HashData> class HashMap
{
typedef pair<HashKey, HashData> HashNode;
public:
    HashMap(size_t b) : _numBuckets(0), _buckets(0) { init(b); }
    void init(size_t b) { _numBuckets = b; _buckets = new vector<HashNode>[b]; }
    bool insert(const HashKey& k, HashData& d)
    {
        size_t b = k();
        for(size_t i = 0, bn=_buckets[b].size(); i<bn; ++i)
            if (_buckets[b][i].first == k)
            {
                d = _buckets[b][i].second;
                return false;
            }
        _buckets[b].push_back(HashNode(k,d));
        return false;
    }
private:
    size_t _numBuckets;
    vector<HashNode>* _buckets;
}
```

Note：I added "d = _buckets[b][i].second;" in the HashMap::insert function.

2. *Explanation*：

- About HashKey：
  i0 and i1 corresponds to the input "literal" id. So I can judge input is inverted or not using (i0/2)*2 != i0. Also, before calling the constructor of HashKey, I will make sure that i0<i1.
  The hashing function (k()) will return the smaller "actual" id, which is i0/2 if both inputs are inverted or both inputs are not inverted. If only one input is inverted, then the hashing function will return the larger "actual" id, which is i1/2. This means that →

  ```
      if( input_1 is inverted XOR input_2 is inverted ) return smaller id;
      else return larger id;
  ```

  Also, only if i0 and i1 is the same in two different HashKey, that two HashKey is identical. (not i0/2 == k.i0/2 && i1/2 ==k.i1/2)
  In this method, every buckets in HashMap will have the same number if numbers of input approximate to infinity.
- About HashMap：
  I will initialize HashMap with _numBuckets = (para[0], which is M in the aag header).

Thus, the return value of the hashing function will never exceed this value and thus there is no need for %(mod), which I simply use size_t b = k() to get the value.
The template I use is <HashKey, unsigned>, in which the unsigned represents the gate id.
I only use the insert function in CirMgr, how I do this will be explained later.

■ Using HashMap in Strashing

1. *Steps*

i. Initialize HashMap to _numBuckets = para[0].
ii. While depth-first-search traversal
iii. Ensure i0 < i1 then hash the gate.
iv. While inserting the gate into hash table, if find identical gate, merge them.
v. Else, simply insert the gate into hash table.

2. *Pseudo-code* ：

```
CirMgr::strash()
{
    HashMap<HashKey, unsigned> hash(para[0]);    //initialize the hash table
    for( every PO )
        depth_first_search_traversal_for_strash(POs, hash); //set mark while traversal
}
```

```
CirMgr::depth_first_search_traversal_for_strash( CirGate*& this_gate, HashMap& hash)
{
    for( two input gates of this_gate )
    {
        if( the input gate is valid and not marked)
        {
            set mark on input gate;
            depth_first_search_traversal_for_strash( this_gate → input gate);
        }
    }
    int i0, i1 = this_gate's literal id;
    if( i0, i1 is not valid ) return;
    if( i0 > i1 ) swap(i0, i1);
    unsigned o_id = this_gate → own_id();
    unsigned gid = o_id;
    if( ! hash.insert ( HashKey(i0, i1), gid )    fail    )
        merge_and_rewire(this_gate, gate_list[gid] );
}
```

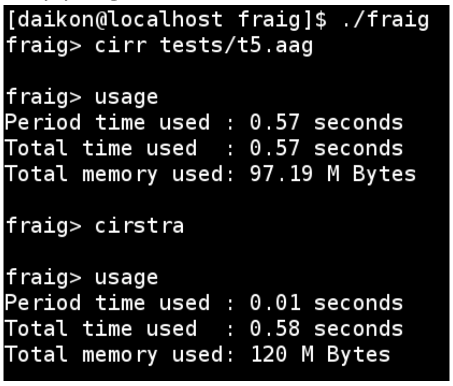
Note：
While inserting, originally gid == o_id, so if the function doesn’t find identical gate, then add this_gate into the hash table. But if the function do find identical gate, the value of gid changes to the id of the gate that is identical to this_gate, then simply merge and rewire these two gates.

3. *Experiments ：(aag file generated by AAGgen)*

i. A huge circuit with few gates required strashing：

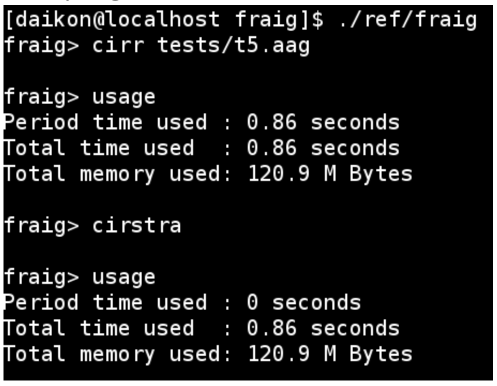
aag 999999 280000 0 1 719999

my program：



```
[daikon@localhost fraig]$ ./fraig
fraig> cirr tests/t5.aag

fraig> usage
Period time used : 0.57 seconds
Total time used  : 0.57 seconds
Total memory used: 97.19 M Bytes

fraig> cirstra

fraig> usage
Period time used : 0.01 seconds
Total time used  : 0.58 seconds
Total memory used: 120 M Bytes
```

ref program：



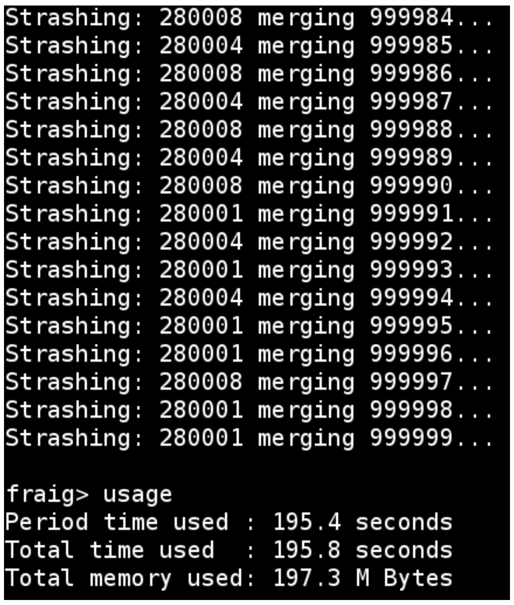
```
[daikon@localhost fraig]$ ./ref/fraig
fraig> cirr tests/t5.aag

fraig> usage
Period time used : 0.86 seconds
Total time used  : 0.86 seconds
Total memory used: 120.9 M Bytes

fraig> cirstra

fraig> usage
Period time used : 0 seconds
Total time used  : 0.86 seconds
Total memory used: 120.9 M Bytes
```

Conclusion：
Basically doesn’t require any time. Although there is no gates to be merge, a hash table is still be constructed in my program. But the ref program seems to create hash table after checking there are actually gates to be merged. This is probably why my program is 0.01 seconds slower.

ii. A huge circuit with most gates required strashing：

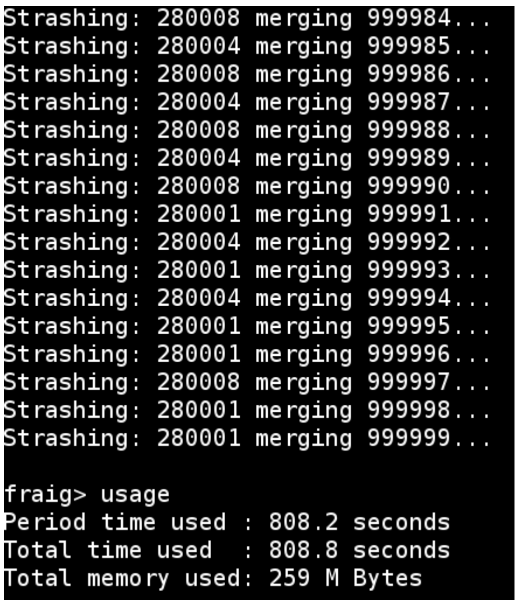
aag 999999 280000 0 999998 719999

my program：



```
Strashing: 280008 merging 999984...
Strashing: 280004 merging 999985...
Strashing: 280008 merging 999986...
Strashing: 280004 merging 999987...
Strashing: 280008 merging 999988...
Strashing: 280004 merging 999989...
Strashing: 280008 merging 999990...
Strashing: 280001 merging 999991...
Strashing: 280004 merging 999992...
Strashing: 280001 merging 999993...
Strashing: 280004 merging 999994...
Strashing: 280001 merging 999995...
Strashing: 280001 merging 999996...
Strashing: 280008 merging 999997...
Strashing: 280001 merging 999998...
Strashing: 280001 merging 999999...

fraig> usage
Period time used : 195.4 seconds
Total time used  : 195.8 seconds
Total memory used: 197.3 M Bytes
```

ref program：



```
Strashing: 280008 merging 999984...
Strashing: 280004 merging 999985...
Strashing: 280008 merging 999986...
Strashing: 280004 merging 999987...
Strashing: 280008 merging 999988...
Strashing: 280004 merging 999989...
Strashing: 280008 merging 999990...
Strashing: 280001 merging 999991...
Strashing: 280004 merging 999992...
Strashing: 280001 merging 999993...
Strashing: 280004 merging 999994...
Strashing: 280001 merging 999995...
Strashing: 280001 merging 999996...
Strashing: 280008 merging 999997...
Strashing: 280001 merging 999998...
Strashing: 280001 merging 999999...

fraig> usage
Period time used : 808.2 seconds
Total time used  : 808.8 seconds
Total memory used: 259 M Bytes
```

Conclusion：

My program is four times faster than ref program.

4. *Performance*：

My program runs faster than ref program, and the larger the circuit is, the greater the advantage of my program over ref program is. The speed of my program is approximately four times faster than ref program.

Simulation：

1. *Steps*：

i. Use depth-first search traversal to go every gate and set the state of the gate. At the same time, hash every AND gate’s id into a temporary vector according to the simulated results.
ii. For every buckets of the temporary vector, sort the gates according to the simulated results.
iii. If neighboring gates has the same simulated results, than add these gates into simt (simulation table).
iv. Renew simt for the first time.
v. Further simulation is done directly on simt.
vi. Renew simt for the first time.
vii. Repeat v to vi for continuous simulation.

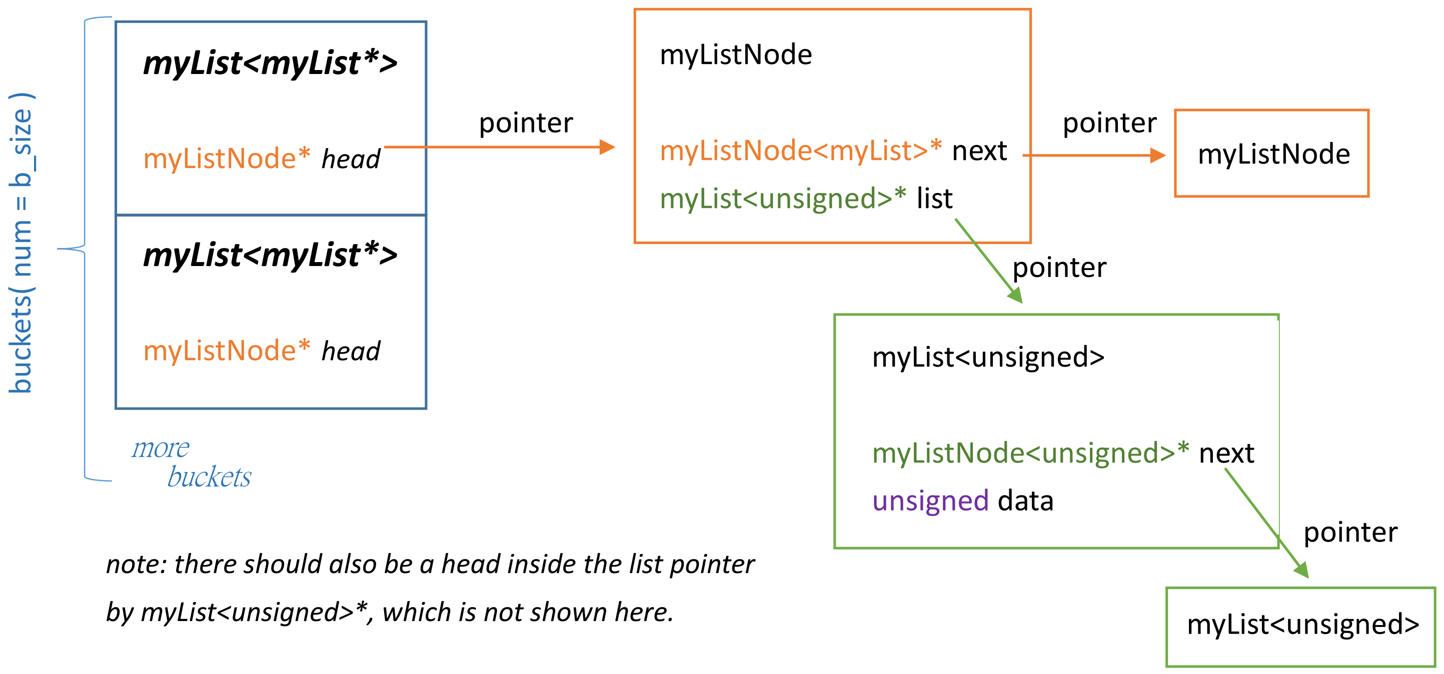
2. *Pseudo-Code:*

```
CirMgr::first_time_simulate()
{
    Simulate_all_gates_by_depth-first-search-traversal();
    vector<unsigned>* v_tmp = vector<unsigned>[b_size];
    For (all gates in DFS list)：
        hashkey = gate’s_state % b_size;
        v_tmp[key].push_back( gate’s_own_id );
    For( every key )
        std::sort_vector( according to gate’s_state );
        If ( there are neighboring gates with the same state )
            Simt[key].insert( these gates );
}
```

```
CirMgr::further_simulation ()
{
    Simulate_all_gates_by_depth-first-search-traversal();
    For_every_buckets_in_simulation_table (simt)
        For_every_list_in_this_bucket
            For_every_gate_in_this_list
                If( gate's_state different from the state of the first gate in the list )
                    Simt[this_key].erase(this_gate);
                    hashkey = gate's_state % b_size;
                    Simt[hashkey].re-insert( gate's state );
    renew_simt();
    If ( too_many_excess_space )
        Refresh_simt();
}
```

```
CirMgr::renew_simt()
{
    for( every_list_in_simt )
    {
        if( the_size_of_list == 1 )
            remove_this_list_from_simt;
        else
            for( every_gate_in_this_list )
                record_gate_address
    }
}
```

3. *Stucture of Simulation Table(simt) (schematic diagram)*



note: there should also be a head inside the list pointer by myList<unsigned>, which is not shown here.*

4. Explanation：

- About Simulation Table(simt):
  I wrote a singly-linked list and put it in myHashMap.h and I use this list for my simt. Every buckets of simt is constructed of a myList< myList<unsigned>* >. This means that inside every buckets, I'm capable of adding several lists and thus adding new FECgroups inside it.
- What is stored in simt：
  Instead of storing the id of the gates or the pointer, I stored the literal id of the gate. The concept is basically the same as how the aag file is written. In this manner, I can handle inverse functionally equivalent candidates (IFECs) by simply add one after id is multiplied by two. For example, if gate number 7 and 8 are IFECs, I may store 15 16 or 14 17 depends on which gate is the first gate in the list (if printFECpairs(), then 7 will be the basis).
- About HashKey：
  My hashkey while simulation is fairly simple, just take the remainder of the gate's state (mod b_size). In order to let the gates spread as equally as possible, my b_size equals to the closest 2's exponential larger than the number of And gates.
- About first_time_simulation()
  The reason I use this method for the first time hashing is for increasing the speed of my program. If I simply hash every gate into simt for the first time, it will results in a numerous quantity of list with only one element (which I have tried before). Then I have to delete it afterwards. So by first_time_simulation(), I would first make sure that this gate is not lonely before inserting it into simt.
- About renew_simt()
  The core of my algorithm is inside renew_simt(). After every time of simulation (either first_time_simulation() or further_simulation()), I will call renew_simt() to renew simt. This means that I will clear every list with only one element and record the "address" of those gates inside a FEC group. "Address" represents the literal id of the first gate in this list. For example, If "literal id " 14,16, and 19 are all in a list (a FEC group) and 14 is the first gate in this list, then address of gates in this list will be 14 (i.e. addr[7]=addr[8]=addr[9]=14).
  During the simulation next time, only the gates with the same address and same state will be inserted into the same list. This allows me to use the whole space of simt for every simulation without ambiguity.
- About refresh_simt()
  After I am sure that my simulation is correct, I started to improve its performance. I discover that after several times of simulation with 64 bits. It is possible that many of the gates are removed and thus the lists inside simt becomes shorter and more sparsely distributed. If the number of list, which is the number of FEC groups, is smaller than 1/40 of buckets number, than I will call refresh_simt() to clear the excess space. The number 1/40 comes up after several experiments of testcase. Inside refresh_simt(), I construct a new simt and assign the myList<unsigned>* inside

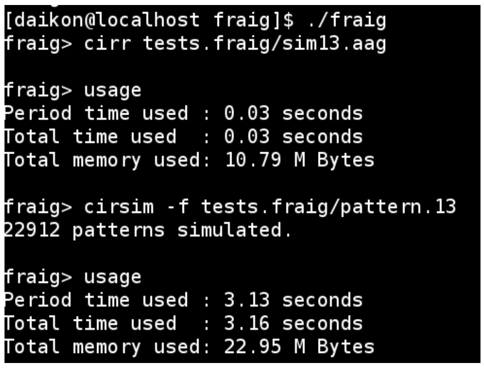
the original simt to the new simt. Delete the old one and assign the new one to simt.

5. *Experiments：*

i. A big circuit with many equivalent gates：

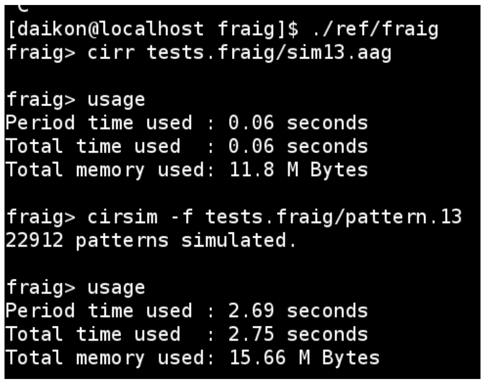
aag 85067 3357 0 3343 81718 (i.e. sim13.aag)

my program：



```
[daikon@localhost fraig]$ ./fraig
fraig> cirr tests.fraig/sim13.aag

fraig> usage
Period time used : 0.03 seconds
Total time used  : 0.03 seconds
Total memory used: 10.79 M Bytes

fraig> cirsim -f tests.fraig/pattern.13
22912 patterns simulated.

fraig> usage
Period time used : 3.13 seconds
Total time used  : 3.16 seconds
Total memory used: 22.95 M Bytes
```

ref program：



```
[daikon@localhost fraig]$ ./ref/fraig
fraig> cirr tests.fraig/sim13.aag

fraig> usage
Period time used : 0.06 seconds
Total time used  : 0.06 seconds
Total memory used: 11.8 M Bytes

fraig> cirsim -f tests.fraig/pattern.13
22912 patterns simulated.

fraig> usage
Period time used : 2.69 seconds
Total time used  : 2.75 seconds
Total memory used: 15.66 M Bytes
```

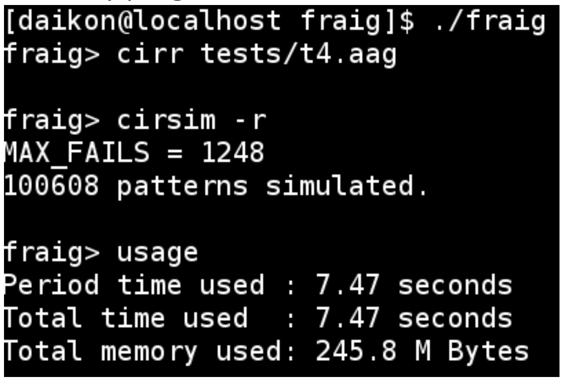
Conclusion：

My program runs slower even when I use singly linked-list and try to refresh the table. I really tried but just can’t be as fast as ref program in this test case. I think a major issue is the method to parse the patterns, because I think I have done a relatively poor job in this part.

ii. A huge circuit with little equivalent gates：(FEC groups about 30 to 50)

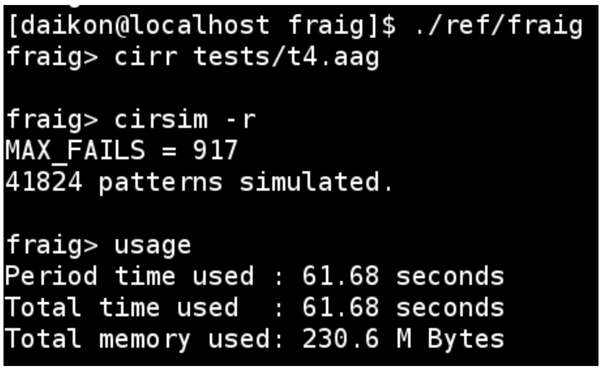
aag 999999 280000 0 33897 799999

my program：



```
[daikon@localhost fraig]$ ./fraig
fraig> cirr tests/t4.aag

fraig> cirsim -r
MAX_FAILS = 1248
100608 patterns simulated.

fraig> usage
Period time used : 7.47 seconds
Total time used  : 7.47 seconds
Total memory used: 245.8 M Bytes
```

ref program：



```
[daikon@localhost fraig]$ ./ref/fraig
fraig> cirr tests/t4.aag

fraig> cirsim -r
MAX_FAILS = 917
41824 patterns simulated.

fraig> usage
Period time used : 61.68 seconds
Total time used  : 61.68 seconds
Total memory used: 230.6 M Bytes
```

Conclusion：

It seems that my program totally defeat ref program, but I have to say, this is because I am tricky. When the users don’t type –o log_file, this means that the users only want to know the FEC groups instead of the states every gates. In suck case, I perform simulation only for those gates inside my simt. In other words, I only give patterns to those PIs who are the fanins of those gates in simt. In this case, the FEC group is quickly reduced to about 100. Compares to the original size of the circuit, doing the trick definitely helps my speed a lot. However, for the above cases, there are too many FEC group, thus the advantage of my trickiness is not apparent.

6. *Performance* :

Strictly speaking, my program runs slower (about 1.2 times slower). However, in cases when the FEC groups are relatively tiny and no need to output log file, my program runs a lot faster.

Fraig :

1. *Steps* :

i. Construct and initialize the SatSolver. Set Var for gates besides PO. Establish the model by depth-first search traversal.
ii. For every list, copy the elements in the list into a vector, and delete this list from simt.
iii. Send this vector to mySolve(), which calls the SatSolver and tries to solve the FEC pairs.
iv. If this group is small, simply let SatSolver to handle. However, if the group is large enough, record the patterns of SAT by SatSolver.
v. If the patterns collected is more than the criterion to re-simulation, re-simulate the group.
vi. Sort the group according to the resulted new state.
vii. If there are neighboring gates in the group with the same state, find an empty bucket in simt and insert these gates into the buckets. (If there are no empty buckets, insert these gates into the middle bucket of simt. However, I have never encounter this problem.)
viii. Continue to solve lists in simt until there are no more left.

2. *Pseudo-Code:*

```
CirMgr::fraig()
{
    SatSolver solver;      solver.initialize();
    For_every_gate_besides_PO
        gate->set_var(solver.newVar());
    Establish_model_by_DFS();
    While_simt_is_not_empty
        For_every_list_in_simt
            List → vector;
            Delete_List_from_simt;
        mySolve( solver, vector );
}
```

```
CirMgr::mySolve ( SatSovler& solver, vector& group )
{
  If ( group size is big ) need_sim = true;
  For_every_pairs_in_group
      Solver.assumeSolve();
      If ( UNSAT )
          merge(pair);
      else if ( SAT )
          if ( need_sim )
          {
              record_patt();
              if( number of patts is enough )
              {
                  simulate();
                  std::sort( according to state );
                  if( neighboring gates has the same state value )
                      find_emtpy_bucket_and_insert ( new FEC gates );
              }
          }
}
```

3. *Explanation*：

- Criterion for re-simulation：

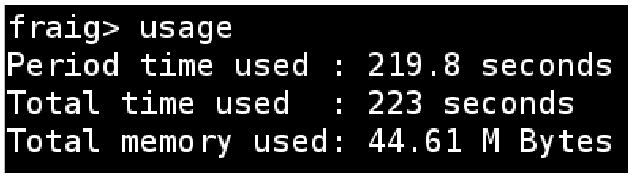
  There are two criterions that I can set on my codes to optimize the speed of my program. The first criterion is the number of gates in a FEC group. The latest version of my program sets this criterion to four. This means that a group with five or more gates needs to be re-simulate "as long as there are enough patterns." Thus, the second criterion is how many patterns should I collect before re-simulate. It is unwise to re-simulate every time a pair is proven SAT, because I have to throw the group back into simt and then parse them into vector to solve. After several tries, I set the second criterion to 6. So if there is 6 patterns collected while solving a group, then re-simulate and throw the group back into simt according to the new resulted state.

4. *Experiments*：

iii. **A big circuit with many equivalent gates**：

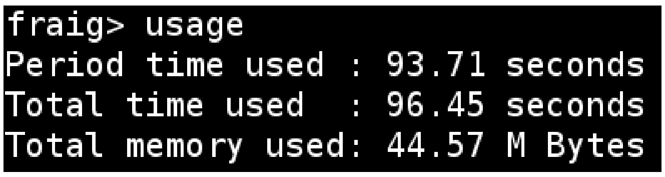
aag 85067 3357 0 3343 81718 (i.e. sim13.aag)

my program：



```
fraig> usage
Period time used : 219.8 seconds
Total time used  : 223 seconds
Total memory used: 44.61 M Bytes
```

ref program：



```
fraig> usage
Period time used : 93.71 seconds
Total time used  : 96.45 seconds
Total memory used: 44.57 M Bytes
```

Conclusion：
My program runs more than two times slower than ref program. However I resulted in a more simplified circuit. (I didn't jump any clauses)

5. *Performance：*

From the onset, I knew my program would run quite slow compared to ref program. And I'm right. This is obvious because I have to parse my list into vector and solve it. While solving, it is possible that I throw the vector back into simt. For a huge group, these actions back and forth costs a lot of time. So I have never expect my program to run faster than ref program. If I have more time, I would try to write a version that I am able to solve a group inside a list, not a vector.

➢ Thoughts：

After I basically finish all the functions of the final project, I view back and found out that I have spend too much time on simulation. This results in the poor performance of my fraig. However, I can't blame anyone about it. I changed the data structure of simulation table from vector to list. And I add the sorting algorithm also to improve the speed. There are various of optimization I try to do on simulation. I did came up with a better program, but I sacrifice the time to write fraig function. When I finally decide to implement fraig, it's a bit too late, because I found out that fraig isn't that simple as I thought. I originally thought that the most difficult part in fraig is to learn how to use SatSovler. But I'm totally wrong. The most difficult part is to re-simulate in an efficient manner. I spent only two to three hours to finish with a fraig function without re-simulation. Then I stuck at simulation for a long time, eventually gave up and choose the easier but duller path.
However, besides the frustration I have had on writing fraig, I did enjoy the process of finishing this program. I learnt a lot and will never forget this particular experience. I felt pretty proud of myself that I come through the course the whole semester and are going to finish here, even though there are lots of improvements to be made on my codes and my coding ability.

➢ Future Improvements：

If I have more time, I would like to arrange my code. Honestly, I have written an bunch of ugly codes in this final project. If I want to further improve the efficiency of my program, I have to first revise those ugly codes. Especially in simulation and fraig, which are so large and lengthy that almost exceed my debugging ability. Also, I would love to try writing the miter function, which is definitely arduous for me, but will be pretty interesting, at least I think.