計算機輔助設計特論 - B101016 曹寓恆 HW1

1. 將所有 On 讀入 OnSet 以及把 Dontcare 讀入 DoncareSet 中，接著用這些資料建立 Minterms，透過 bitCount()算出每個數分別有幾個 1，將他們分成”變數數量+1”組。

```
ReadFile_and_Initial(inFile, Variables, OnSet, DontCareSet);
vector<vector<Term>> Minterms(Variables + 1);
SetMinterms(Variables, OnSet, DontCareSet, Minterms);
```

2. 透過 while(!minterms.empty())，作為結束找 Prime implicant 的依據，PairInGroups 是將 Minterms 中相鄰的兩組，也就是 1 的個數相差為 1 的資料一一配對嘗試化簡。

```
vector<Term> PrimeImplicant;
QM qm = {Variables, OnSet, Minterms, PrimeImplicant};
while (!Minterms.empty())
{
    int size = Minterms.size();
    for (int i = 0; i < size - 1; i++)
        qm.PairInGroups(Minterms[i], Minterms[i + 1]);
    qm.ProcessLastGroup();
}
```

3. 這是 **PairInGroups** 的程式碼，一開始 **CanMerge()**先判斷一組數能否化簡(同時回傳 **PI_chart** 以及 **Merge** 結果)，能的話就將這組數打勾表示已經有配對到可以刪掉了，接下來檢查 **NotExist()**化簡後的結果有沒有重複出現，若沒出現過就放進 **Group1** 等待下一輪化簡。在 **Group1** 的第 **0** 個數和 **Group2** 中所有人都配對過且無法化簡表示他是 **Prime implicant** 計算完畢，最後將 **Group1** 第 **0** 個數刪掉。

```
    for (int i = 0; i < G1_Size; i++)
        {
            int j = 0;
            for (auto &it : Group2)
            {
                Term MergeResult;
                if (CanMerge(Group1[0], Group2[j], MergeResult))
                {
                    Group1[0].mark = true;
                    Group2[j].mark = true;
                    if (NotExistInGroup(Group1, MergeResult))
                        Group1.push_back(MergeResult);
                }
                j++;
            }
            if (!Group1[0].mark)
                PutInPrimeImplicant(Group1[0]);
            Group1.erase(Group1.begin());
        }
```

4. 配對期間會把已經計算完畢的數丟掉，所以一輪 **while()**中配對後有些 **Group** 會變空，**ProcessLastGroup** 是一輪配對的收尾。第一步把最尾端的 **Group** 的元素依據打勾情形放進 **Prime implicant** 中或是直接刪掉。最後，檢查 **Mimterms** 中哪些 **Group** 空了就刪掉，結束一輪 **while()**，當所有 **Minterms** 都計算完了 **Minterms.empty()**就會成立，離開 **while()**，結束 **Quine Mcluskey**。

```
void ProcessLastGroup()
{
        while (!Minterms.back().empty())
        {
            if (!Minterms.back()[0].mark)
                PutInPrimeImplicant(Minterms.back()[0]);
            Minterms.back().erase(Minterms.back().begin());
        }
        Minterms.pop_back();
        int size = Minterms.size();
        for (int i = 0; i < size;)
        {
            if (Minterms[i].empty())
            {
                Minterms.erase(Minterms.begin() + i);
                size--;
            }
            else
                i++;
        }
    }
```

5. 產生 **implicant.txt**

```
set<string> MinCover;
    Output out = {Variables, PrimeImplicant, MinCover,inFile, outFile1,
outFile2};
    out.OutputImplicant();
```

6. **FindMinCover()**是 **Petrick's method** 的主程式，由於乘開後會是 **P1P3+P2P4P7…** 的型態，**vector<set<string>> P1P2_Set** 就是用來儲存乘開的結果。**set<string> P1P2** 用來儲存即將要和 **P1P2_Set** 相乘展開的項(後面一項)，例如: **(P1P2P3+P4P7)(P5+P6)**，下圖迴圈是將第一項設為 **P1P2_Set** 的初始值

```
set<string> FindMinCover()
    {
        vector<set<string>> P1P2_Set;
        auto itO = OnSet.begin();
        for (const auto &itP : PrimeImplicant)
        {
            bool flag = false;
            set<string> P1P2;
            if (itP.PI_chart[itO->second])
            {
                flag = true;
                P1P2.insert(itP.value);
            }
            if (flag)
                P1P2_Set.push_back(P1P2);
        }
```

7. 下圖是 **FindMinCover()**的下半段，上面的迴圈不斷呼叫 **Merge()**來將兩項合併**(**細節在 **8.9.)**，全部合併結束後呼叫 **CountLiteral** 找到最少 **literal** 的一組 **Cover** 並回傳給 **MinCover**，接著 **main()**中呼叫 **OutputMinCover** 產生 **output.txt** 全部程式執行完畢

```
        itO++;
        for (; itO != OnSet.end(); itO++)
        {
            set<string> P1P2;
            for (const auto &itP : PrimeImplicant)
                if (itP.PI_chart[itO->second])
                    P1P2.insert(itP.value);
            P1P2_Set = Merge(P1P2_Set, P1P2);
        }
        int MinLiteral = INT_MAX, i = 0, MinIndex = 0;
        for (auto &it : P1P2_Set)
        {
            int literal = CountLiteral(it);
            if (MinLiteral > literal)
            {
                MinLiteral = literal;
                MinIndex = i;
            }
            i++;
        }
        return P1P2_Set[MinIndex];
    }
};
```

8. 乘上，**Merge()**中把 **P1P2_Set** 和 **P1P2** 乘開，利用 **set** 元素不重複的特性實現 **XXY = XY** 的吸收律，乘開的結果還需要經過 **X+XY=X** 的化簡，這一部分呼叫 **Absrption()**處理

```
vector<set<string>> Merge(vector<set<string>> &P1P2_Set, set<string>
&P1P2)
    {
        vector<set<string>> MergeResult;
        for (const auto &itS : P1P2_Set)
        {
            for (const auto &it2 : P1P2)
            {
                set<string> p1p2;
                p1p2.insert(it2);
                for (const auto &it1 : itS)
                    p1p2.insert(it1);
                MergeResult.push_back(p1p2);
            }
        }
        Absorption(MergeResult, P1P2);
        return MergeResult;
    }
```

9. Absorption()先將 **vector** 依包含變數數量排序，例如 **P1 + P2P3 + P1P5P6…**，以 **P1** 開始，在後面的 **P2P3** 等尋找裡面是否包含，若包含則刪掉，例如 **P1P5P6** 包含 **P1**。但是這種方法沒效率，若能想到樹的比較大小依據來建立 **set<set<string>>**取代 **vector<set<string>>**效率應能提升不少。以上是一輪 **Merge()**的流程，不斷 **Merge()**到最後 **P1P2_Set.size() == 1**，**POS** 展開完畢

```
void Absorption(vector<set<string>> &MergeResult, set<string> &P1P2)
    {
        sort(MergeResult.begin(), MergeResult.end(), sortP1P2);
        int size = MergeResult.size();
        for (int i = 0; i < size - 1; i++)
        {
            for (int j = i + 1; j < size;)
            {
                int count = 0;
                for (auto itI = MergeResult[i].begin(); itI != 
MergeResult[i].end(); itI++)
                {
                    if (MergeResult[j].find(*itI) == 
MergeResult[j].end())
                        break;
                    else
                        count++;
                }
                if (count == MergeResult[i].size())
                {
                    MergeResult.erase(MergeResult.begin() + j);
                    size--;
                }
                else
                    j++;
            }
```