

# Implementation of Karnaugh map

This file is written in Traditional Chinese, encoding in UTF-8.

此程式用於實現輸出 Karnaugh map 的圖示、prime implicants、essential prime implicants 及 variables 的 minimum minterm expansion。最多支持 4 個變數的輸入。若有超過一組 minimum minterm expansion 時，只會輸出其中一項。輸入檔案名稱為 input.txt，輸出檔案名稱為 output.txt。

## 壓縮檔中的程式碼

- `kmap.java`：主程式。
- `qmc.java`：實現 Quine-McCluskey Algorithm。
- `petrick.java`：實現 Petrick's method。
- `FileIO.java`：實作讀取 `.txt` 檔案的功能。

## 撰寫環境

- Java 17 Edition
- Windows 10

## qmc.java 中的 Methods

程式參照 [Quine–McCluskey algorithm](#) 撰寫。

- `getLetters()`：依照輸入變數數量提供英文字母（例：4 個變數輸出 a, b, c, d）。
- `decimalToBinary()`：將輸入的十進位數字轉換為二進位。
- `fillUp()`：將轉換為二進位的數字依照變數數量補零（例：4 轉換為二進位為 10，若變數數量為 4 則補零成 0010）。
- `isGreyCode()`：檢查兩二進位數字是否為 [Grey code](#)。
- `replaceCompliment()`：與 `isGreyCode()` 搭配使用，將兩二進位數字相異的位元替換為 -。
- `isArray()`：檢查一字串是否在一 [ArrayList](#) 中。
- `reduce()`：將各二進位數字依照 Quine–McCluskey algorithm 合併。
- `getValue()`：將二進位數字以 [minterms](#) 轉換為英文字母。
- `isArrayEqual()`：檢查兩 [ArrayList](#) 是否相同。

## petrick.java 中的 Methods

程式參照 [Petrick's method](#) 撰寫。

- `match()`：尋找能蘊含一 minterm 的所有 prime implicants。
- `checkPI()`：檢查一 minterm 是否能以 minterm expansion 中的任一 prime implicant 表示。
- `isEPI()`：若 `match()` 只尋找到一個 prime implicant，則該 prime implicant 為 [essential prime implicant](#)。
- `reduceEPI()`：將重複的 essential prime implicants 除去。
- `combine()`：將 [product-of-sums](#) 轉換為 [sum-of-products](#)。
- `split()`：將 sum-of-products 中的每項 product 提取出來。
- `count()`：與 `split()` 搭配使用，用於選出擁有最少 prime implicants 的 product。

## kmap.java 中的 Methods

- `readFile()`：讀取 input.txt。
- `split()`：將檔案內容中的字元分離。
- `strToInt()`：將字串中資料型態為 `String` 的數字轉換為 `int`。
- `init()`：實作 `readFile()`、`split()`、`strToInt()`。
- `fileCreator()`：若資料夾中已有存在的 output.txt，則清空其內容。
- `printer()`：輸出內容至 output.txt 中。
- `replacer()`：於 `printer()` 中使用，用於輸出 Karnaugh map 的圖示。

## input.txt 格式

```
v 4
m 0,4,5,10,11
d 1,13,14,15
```

- 第一列為變數數量，範圍為 0 到 4。
- 第二列為 minterms，範圍為 0 到  $2^n - 1$ 。
- 第三列為 don't cares，範圍同 minterms。

## output.txt 格式

Karnaugh map 圖示以記事本顯示為標準。  
此處以 4 個變數為例。

```
\AB|      |
CD \ | 00 01 11 10|
-----+---+---+---+
    00| 1 | 1 | 0 | 0 |
-----+---+---+---+
    01| x | 1 | x | 0 |
-----+---+---+---+
    11| 0 | 0 | x | 1 |
-----+---+---+---+
    10| 0 | 0 | x | 1 |
-----+---+---+---+
```

Prime implicant:a'c',ac,bc'd  
Essential prime implicant:a'c',ac  
F(A,B,C,D)=a'c'+ac

- 上方為 Karnaugh map。
- 文字的第一列為 prime implicants。
- 文字的第二列為 essential prime implicants。
- 文字的第三列為 minimum minterm expansion。

## 主程式實作

### Step 1-1

```
int variableNumbers = init(0)[0];
if(variableNumbers > 4){
    System.out.println("The max variable number is 4!");
    return;
}

qmc qmc = new qmc(variableNumbers);

ArrayList<String> mintermExpansion = new ArrayList<>();
ArrayList<String> realMinterms = new ArrayList<>(); //realMinTerms doesn't consist of don't cares.

int m, d;
//adding minterms to minterm arraylist.
for(int i = 0; i < init(1).length; i++){
    m = init(1)[i];
    realMinterms.add(qmc.fillUp(qmc.decimalToBinary(m)));
    mintermExpansion.add(qmc.fillUp(qmc.decimalToBinary(m)));
}
//adding dontcares to minterm arraylist. (in sorting and reducing phase, we see dontcares as minterms.)
for(int i = 0; i < init(2).length; i++){
    d = init(2)[i];
    mintermExpansion.add(qmc.fillUp(qmc.decimalToBinary(d)));
}

Collections.sort(mintermExpansion);
Collections.sort(realMinterms);
```

- 取得變數數量、minterms 與 don't cares。
- 注意到此處將 don't cares 當作 minterms，並創建 `realMinterms`，用以存放 minterms。

### Step 1-2

```

do{
    mintermExpansion = qmc.reduce(mintermExpansion);
    Collections.sort(mintermExpansion);
}
while(!qmc.isArrayEqual(mintermExpansion, qmc.reduce(mintermExpansion)));

```

- 利用 `reduce()`、`minterms` 與 `don't cares` 輸出 `minterm expansion`。

Quine-McCluskey Algorithm 到此處結束。

## Step 2-1

```

for(int i = 0; i < realMinterms.size(); i++){
    ArrayList<String> pn = new ArrayList<>();
    pn = petrick.match(realMinterms.get(i), mintermExpansion);
    if(petrick.isEPI(pn)){
        EPI.add(pn.get(0));
    }
    POS.add(pn);
}

```

- 利用 `match()` 與 `isEPI()` 尋找能蘊含一 `minterm` 的所有 `prime implicants`。尋找完所有 `minterms` 時組合成 `product-of-sums`。
- 同時尋找 `essential prime implicants`。

## Step 2-2

```

ArrayList<String> SOP = new ArrayList<>(); //sum-of-products
while(POS.size() > 1){
    POS.add(petrick.combine(POS.get(0), POS.get(1)));
    POS.remove(0);
    POS.remove(0);
}
SOP = POS.get(0);

```

- 利用 `combine()` 與 `ArrayList` 中，新資料會置於陣列最後方的特性，將 `product-of-sums` 轉換為 `sum-of-products`。
- 注意到 `combine()` 中已處理 `XX = X` 的情況。

## Step 2-3

```

ArrayList<ArrayList<String> > mintermExpansionFinal = new ArrayList<>();
int counter = petrick.count(petrick.split(SOP.get(0), variableNumbers));
for(int i = 0; i < SOP.size() - 1; i++){
    int a = petrick.count(petrick.split(SOP.get(i), variableNumbers));
    if(a < counter){
        counter = a;
    }
}
for(int i = 0; i < SOP.size(); i++){
    int a = petrick.count(petrick.split(SOP.get(i), variableNumbers));
    if(a == counter){
        mintermExpansionFinal.add(petrick.split(SOP.get(i), variableNumbers));
    }
}

```

- 尋找 `prime implicants` 最少的 `sum-of-products`。

## Step 2-4

```

EPI = petrick.reduceEPI(EPI);
ArrayList<String> PI = mintermExpansionFinal.get(0); //prime implicants
ArrayList<String> minimumMinterm = new ArrayList<>(); //minimum minterm expansion

ArrayList<String> temp = new ArrayList<>();
for(int i = 0; i < realMinterms.size(); i++){
    if(petrick.checkPI(realMinterms.get(i), EPI)){
        temp.add(realMinterms.get(i));
    }
}
//if all EPIs consist of all indexes, then the minimum minterm is the sop of EPIs.
if(qmc.isArrayEqual(temp, realMinterms)){
    minimumMinterm = EPI;
}
else{
    minimumMinterm = PI;
}

```

- 利用 reduceEPI() 將重複的 essential prime implicants 除去。
- 利用 checkPI() 檢查所有 essential prime implicants 是否蘊含所有 minterms，若是，則 minimum minterm expansion 為所有 essential prime implicants 的組合；否則 minimum minterm expansion 為所有 prime implicants 的組合。

## Step 2-5

```

Collections.sort(PI);
Collections.sort(EPI);
Collections.sort(minimumMinterm);
printer(init(1), init(2), variableNumbers, qmc, PI, EPI, minimumMinterm);

```

- 排序後輸出。

程式到此處結束。

## 測試結果

### Testcase 1

```

\AB\
CD \ | 00  01  11  10|
-----+---+---+---+---+
00| 1 | 1 | 0 | 0 |
-----+---+---+---+
01| x | 1 | x | 0 |
-----+---+---+---+
11| 0 | 0 | x | 1 |
-----+---+---+---+
10| 0 | 0 | x | 1 |
-----+---+---+---+

Prime implicant:bc'd,a'c',ac
Essential prime implicant:a'c',ac
F(A,B,C,D)=a'c'+ac

```

第 1 列, 100% Unix (LF) UTF-8

## Testcase 2

```
output.txt - 記事本
檔案(F) 編輯(E) 格式(O) 檢視(V) 說明
\AB|
C \ | 00 01 11 10|
-----+-----+-----+
    0| 1 | 1 | 0 | 0 |
-----+-----+-----+
    1| x | 1 | x | 0 |
-----+-----+-----+

Prime implicant:bc,a'
Essential prime implicant:a'
F(A,B,C)=a'
```

## Testcase 3

```
output.txt - 記事本
檔案(F) 編輯(E) 格式(O) 檢視(V) 說明
\ A|
B \ | 0 1|
-----+-----+
    0| 1 | 0 |
-----+-----+
    1| x | 0 |
-----+-----+

Prime implicant:a'
Essential prime implicant:a'
F(A,B)=a'
```

## References

- [AkshayRaman/Quine-McCluskey-algorithm](#)