

Problem statement

1)프로그램의 목적

1. 퀸 맥클러스키 방법을 활용하여 prime implicant와 essential prime implicant를 찾고 논리식을 간소화하여 SOP를 찾고 그 SOP의 최소 cost를 찾는 프로그램을 만든다.

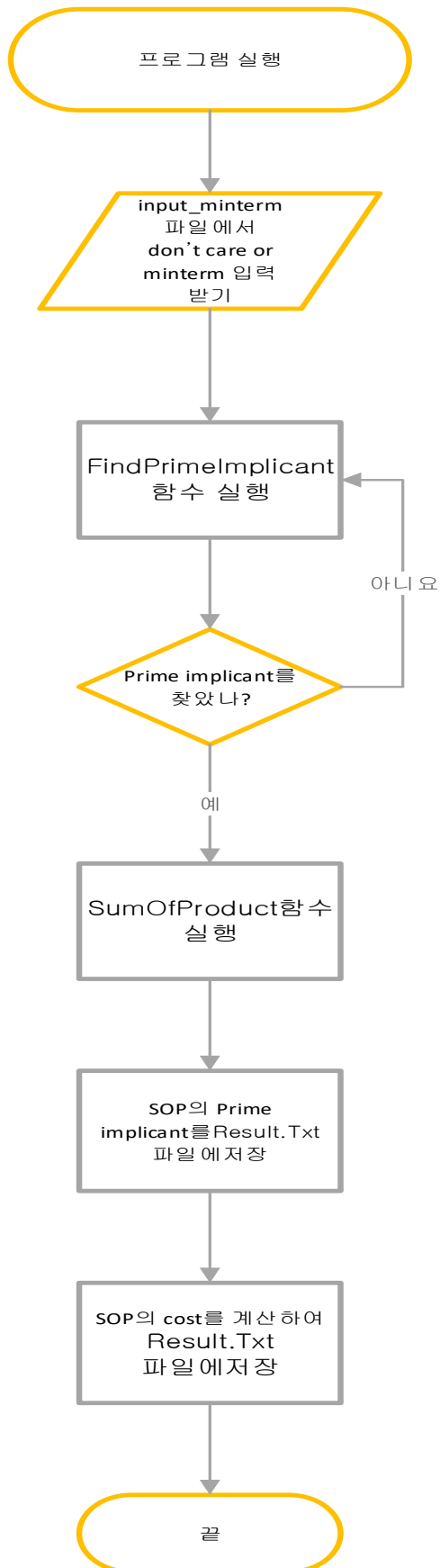
2)프로그램 기능

1. 퀸 맥클러스키 방법을 활용하여 prime implicant를 찾는다.
2. prime implicant 들을 가지고 essential prime implicant을 찾는다.(이 기능의 알고리즘을 구상하기 위해 노력을 했지만 끝내 구현 하지 못 하였습니다..)
3. essential prime implicant와 prime implicant을 이용하여 SOP를 찾고 그SOP의 cost를 계산한다. (위 프로그램은 essential prime implicant을 찾지 못하여 prime implicant만 가지고 SOP를 구성하여 cost를 계산합니다.)

3)프로그램 예외처리

1. input_minterm.txt 파일을 열지 못한 경우. error메시지를 출력하고 프로그램을 종료한다.
2. input_minterm.txt에서 input의 개수를 입력하지 않은 경우. error메시지를 출력하고 프로그램을 종료한다.
3. result.txt 파일을 열지 못한 경우. Error 메시지를 출력하고 프로그램을 종료한다.
4. 예를 들어 input의 개수가 4개이면 d XXXX혹은 m XXXX의 형식을 만족해야 하는데 만족하지 못한 경우 error메시지를 출력하고 프로그램을 종료한다.

flow chart



주요함수 pseudo code

1) FindPrimeImplicant (퀵 맥클러스키 방법에서 prime Implicant를 찾는 함수)

```
int FindPrimeImplicant(char ** InputMinterm, int input, char **Tureminterm)
{
    for (int i = 0; i < 비교할 항의 개수; i++)
    {
        for (int j = 0; j < 비교할 항의 개수; j++)
        {
            for (int k = 0; k < input의 개수; k++)
            {
                if (InputMinterm[i][k] != InputMinterm[j][k])
                {
                    해밍디스턴스 차이를 확인한다.
                }
            }
            if (해밍디스턴스가 1인경우)
            {
                1차이나는 부분을 '_'으로 저장하고
                따로 배열에 저장.
            }
        }
        if (해밍 디스턴스차가 0인경우)
        {
            /*Prime implicant를 찾은 경우*/
            따로 배열에 저장
        }
    }
    if (해밍디스턴스차이가 1인 경우가 없는 경우)
    {
        /*해밍디스턴스차이가 1인 경우가 없는 경우 그 배열에 있는 모든 문자열들은
        prime implicant입니다.*/
        SumOfProduct함수를 실행시킵니다.
        0을 반환합니다. }
    if (해밍 디스턴스가 1인 경우가있는 경우)
    {
        /*(프라임 임플리칸트들만 있는 문자열을 못 찾은 경우)
        FindPrimeImplicant함수 재실행.}
    }
}
```

2) SumOfProduct (Prime implicant를 통해 sum of product를 구하는 함수이며 SOP의 cost를 구할 수 있는 함수이다.)

int SumOfProduct()인자로 FindPrimeImplicant함수에서 찾은 prime implicant를 모아둔 배열과 prime implicant의 개수, input의 개수를 받는다.

```
{
    아웃풋스트림 클래스 변수 fout을 선언한다.
    fout을 통해 result.txt"파일을 연다.
    prime implicant 배열을 result.txt파일에 한줄씩 저장.
    /*아래는 SOP의 cost를 계산하는 과정이다.*/
    변수 cost선언
    for (int i = 0; i < prime implicant의 개수; i++)
    {
        for (int j = 0; j < input; j++)
        {
            if (prime implicant에 '0'이 있으면)
            {
                cost += 2;
                NAND게이트의 입력개수 증가;
            }
            else if (prime implicant에 '1'이 있으면)
            {
                NAND게이트의 입력개수 증가;
                cost수 변화없음;
            }
        }
    }
    if (NAND게이트의 입력 개수가 한개보다 많을때)
    {
        /*NAND게이트의 인풋의 갯수에 두배로
        트랜지스터의 개수(예를들어 2인풋인 경우 트랜지스터 4개 3인풋인경우
        트랜지스터 6개)가 늘어남*/
        cost=NAND게이트의 입력 개수*2;
    }
    if (NAND게이트의 입력 개수가 한개보다 많을 때)
    {
        //우리가 구하고자하는 것은 AND게이트임으로 NOT게이트(트랜지스터2
        개)를 추가하여야한다.
```

```

        cost+=2;
    }

}

if (prime implicant 개수가 1개보다 많을때)
{
    /*프라임 임플리칸트의 갯수가 NOR게이트의 인풋의 갯수가 됨으로 인풋의 갯수
    에 두배로
    트랜지스터의 개수(예를들어 2인풋인 경우 트랜지스터 4개 3인풋인경우 트랜지
    스터 6개)가 늘어난다*/
    cost= prime implicant 개수*2
}

if (termnumber > 1)
{
    //우리가 구하고자하는 것은 OR게이트임으로 NOT게이트(트랜지스터2개)를 추가
    하여야한다.
    cost+=2;
}

cost를 파일에 저장;
}

```

Verification strategy

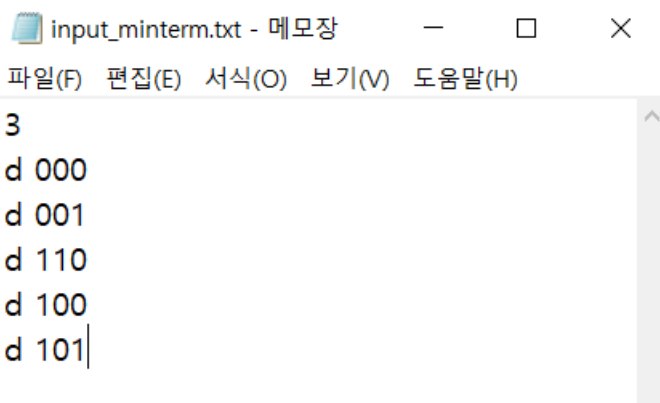
검증 전략

1. Input_minterm로부터 정확하게 minterm들을 읽어와야한다.
2. 프로그램의 결과로 나온 implicant은 prime implicant이어야 한다.
3. 프로그램은 SOP에 대한 적절한 cost를 계산하여야 한다.
4. 결과를 result.txt파일에 정확히 입력 해야 한다.

corresponding examples

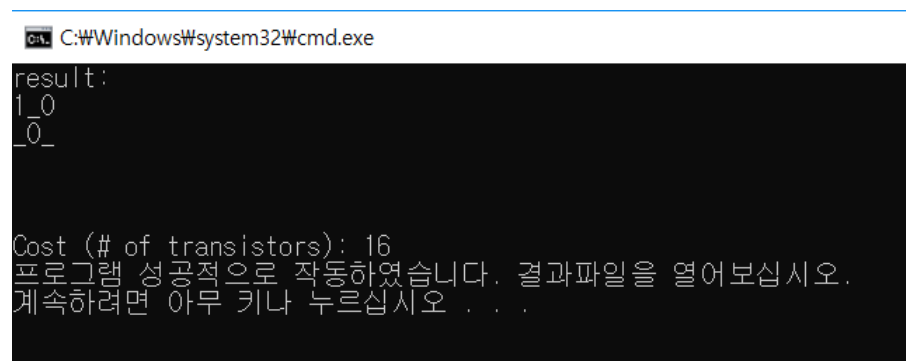
첫번째 예시

Input_minterm파일

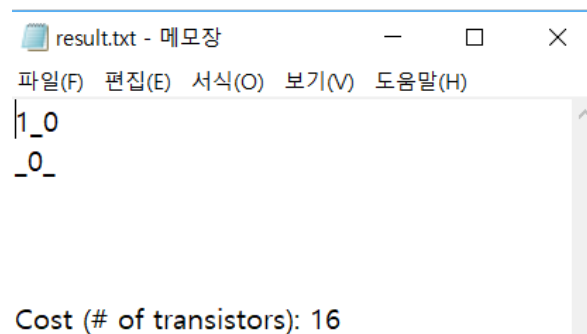


일 경우

프로그램 결과 하면



파일 결과 화면



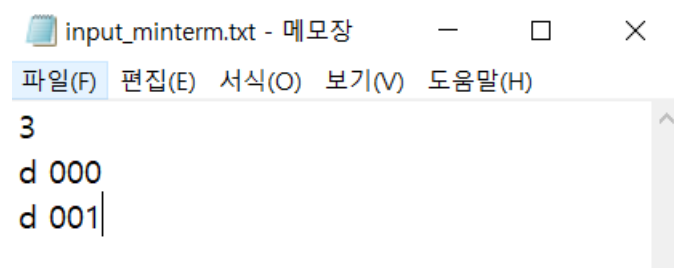
각각의 1_0과 _0_을 $AC'+B'$ 라 할 수 있고 각각은 prime implicant이며

2개의 인버터(cost 4개), 2인풋AND게이트1개(cost 6개), 2인풋OR게이트1개(cost 6개)가 필요하여

총 cost는 16이고 알맞은 결과가 나온다.

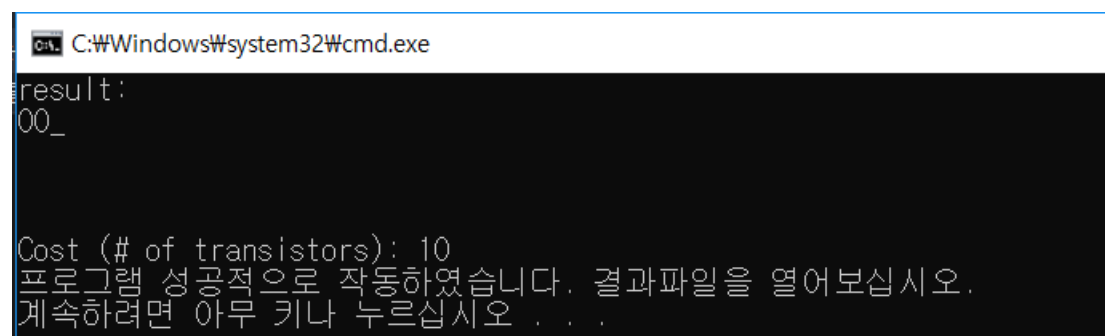
두번째 예시

Input_minterm파일

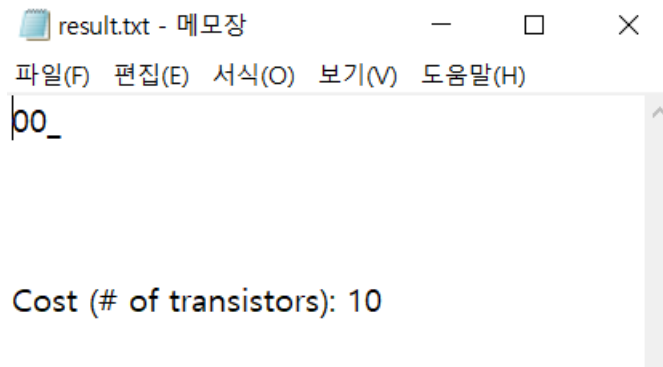


일 경우

프로그램 결과 하면



파일 결과 화면



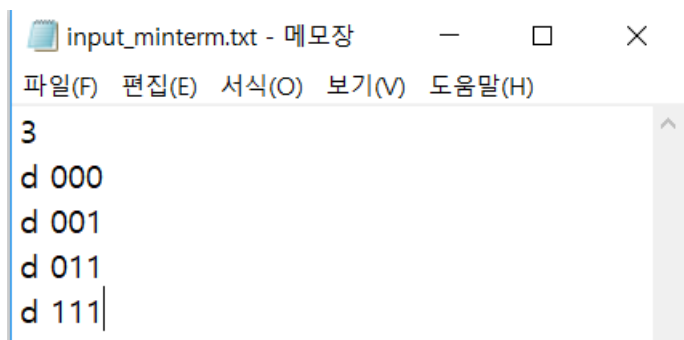
00_을 A'B'라 할 수 있고 prime implicant이며

2개의 인버터(cost 4개), 2인풋AND게이트1개(cost 6개) 필요하여

총 cost는 10이고 알맞은 결과가 나온다.

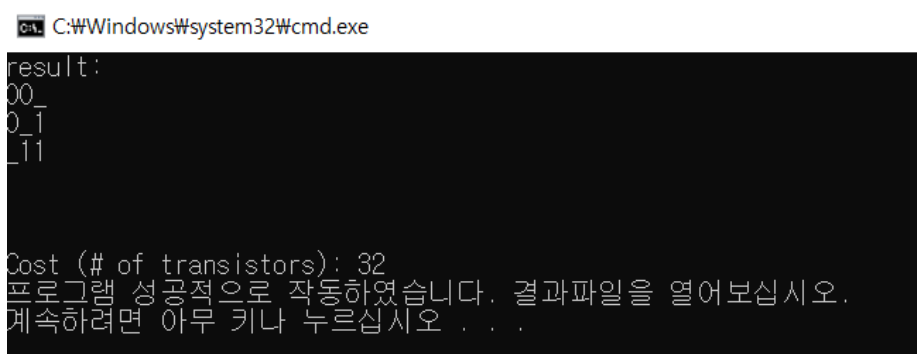
세번째 예시

Input_minterm파일

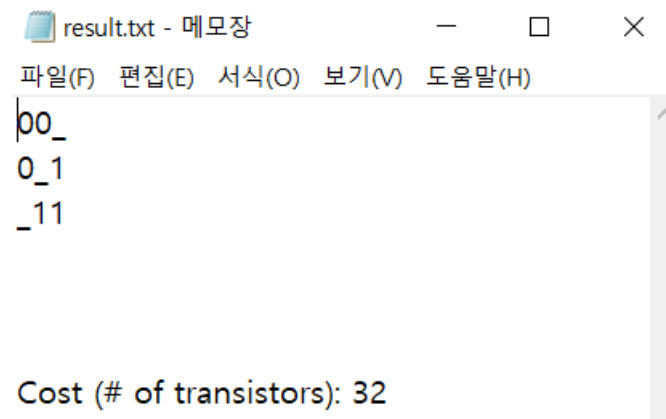


일 경우

프로그램 결과 하면



파일 결과 화면



```
result.txt - 메모장
파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)
00_
0_1
_11

Cost (# of transistors): 32
```

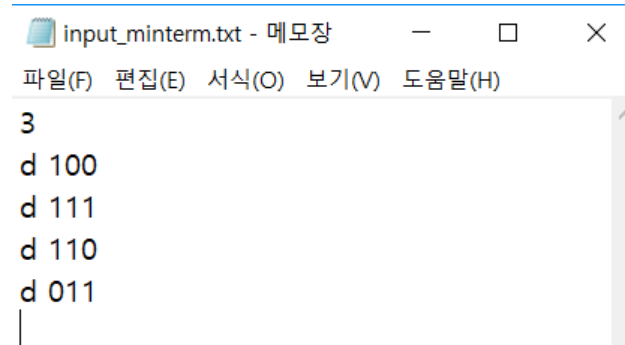
$A'B' + A'C + BC$ 라 할 수 있고 각각은 prime implicant이며

3개의 인버터(cost 6개), 2인풋AND게이트3개(cost 18개), 3인풋 OR게이트1개(cost 8개)필요하여

총 cost는 32이고 알맞은 결과가 나온다.

네번째 예시

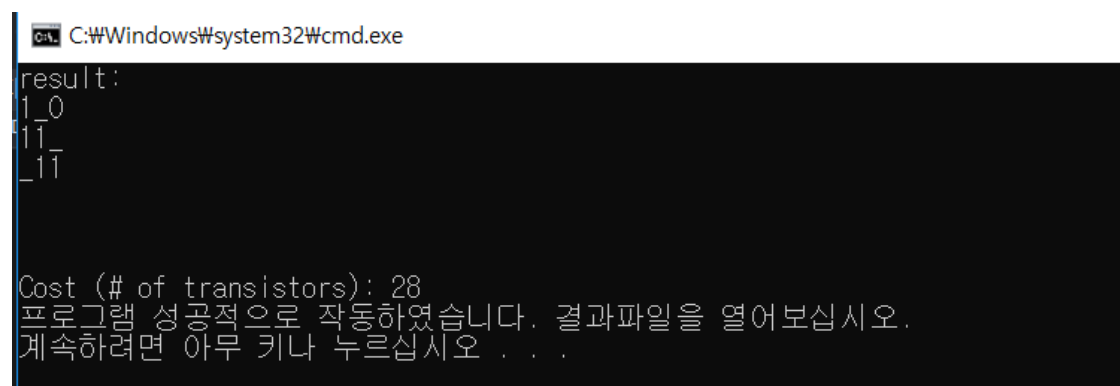
Input_minterm파일



```
input_minterm.txt - 메모장
파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)
3
d 100
d 111
d 110
d 011
|
```

일 경우

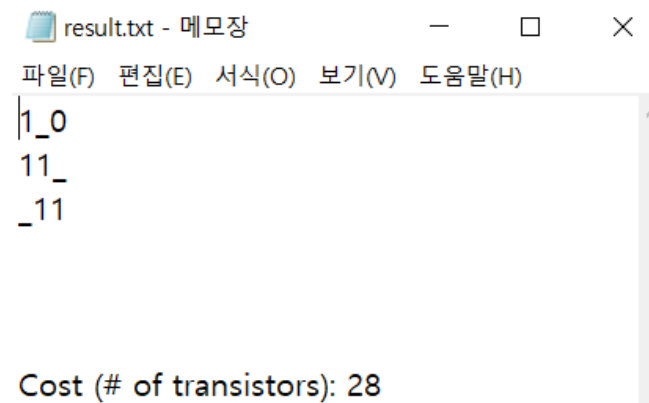
프로그램 결과 하면



```
C:\Windows\system32\cmd.exe
result:
i_0
i_1
_i_1

Cost (# of transistors): 28
프로그램 성공적으로 작동하였습니다. 결과파일을 열어보십시오.
계속하려면 아무 키나 누르십시오 . . .
```

파일 결과 화면



```
1_0
11_
_11

Cost (# of transistors): 28
```

$A'C + AB + BC$ 라 할 수 있고 각각은 prime implicant이며

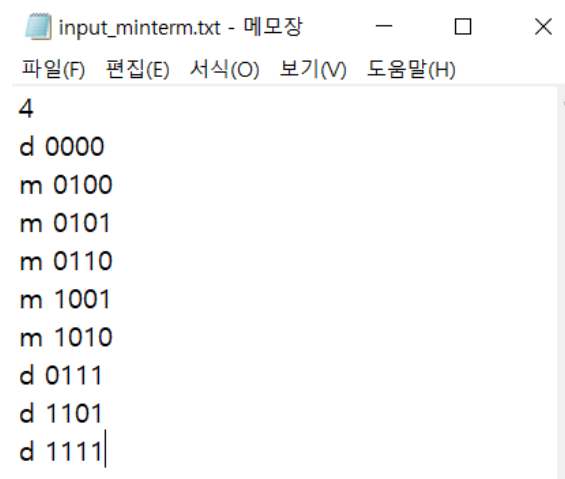
1개의 인버터(cost 2개), 2인풋AND게이트3개(cost 18개), 3인풋OR게이트1개(cost 8개)필요하여

총 cost는 28이고 알맞은 결과가 나온다.

A testbench that I think it is very hard to solve

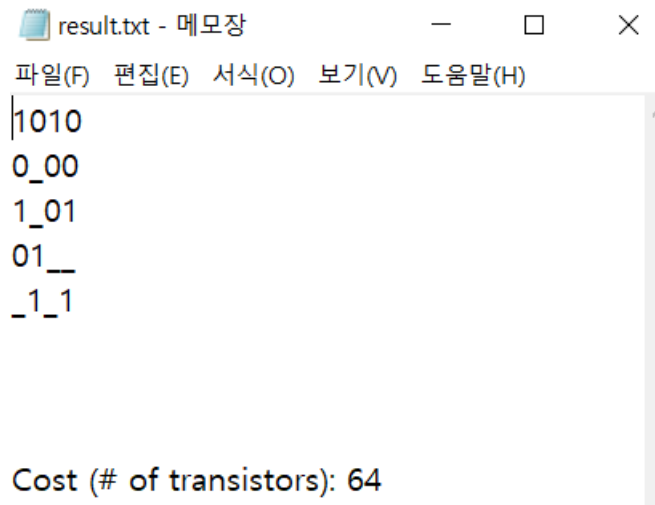
내가 생각하기에 가장 해결하기 어려웠던 테스트 벤치는 아래와 같은 경우이다.

Input_minterm파일이 아래와 같은 경우



```
4
d 0000
m 0100
m 0101
m 0110
m 1001
m 1010
d 0111
d 1101
d 1111
```

프로그램 실행의 결과가 아래와 같이 나오는데



```
result.txt - 메모장
파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)
1010
0_00
1_01
01__
_1_1

Cost (# of transistors): 64
```

SOP를 $AB'CD' + A'C'D' + AC'D + A'B + BD$ 로 나타낼 수 있는데 이는 잘못된 결과이다.

위 식을 다음과 같이 $A'B + AC'D + AB'CD'$ 로 간소화할 수 있기 때문이다. 이러한 결과 나오는 이유는 프로그램에 필수주항을 찾는 과정의 알고리즘을 구현할 수 없었기 때문이다. 이 필수 주항을 찾는 알고리즘을 추가한다면 $AB'CD' + A'C'D' + AC'D + A'B + BD \rightarrow A'B + AC'D + AB'CD'$ 로 결과적으로는 위의 cost보다 더 적은 cost가 나올 것이다.

필수 주항을 찾는 알고리즘을 구현하지 못한 것에 아쉬움을 느끼며 향후 기회가 된다면 필수 주항을 찾는 알고리즘을 추가해 더 간소화된 SOP와 더 적은 cost를 계산하는 프로그램을 만들고 싶다.