Two-level logic is necessary

Some Boolean functions cannot be represented with one logic level
\((\sum f) + (a \land b)\)

Two-level logic is sufficient

\(f(a, b)\)

All Boolean functions can be represented with two logic levels
* Given \(k\) variables, \(2^k\) minterm functions exist
* Select arbitrary union of minterms

Two-level well-understood

* As we will see later, optimal minimization techniques known for two-level
* However, optimal two-level solution may not be optimal solution
  * Sometimes a suboptimal solution to the right problem is better than the optimal solution to the wrong problem

Two-level sometimes impractical

Consider a 4-term XOR (parity) gate:
\(a \oplus b \oplus c \oplus d\)

\[(\sum \bar{a} \bar{b} \bar{c} \bar{d}) + (\sum \bar{a} \bar{b} \bar{c} d) + (\sum \bar{a} \bar{b} c \bar{d}) + (\sum \bar{a} \bar{b} c d) + (\sum a \bar{b} \bar{c} \bar{d}) + (\sum a \bar{b} \bar{c} d) + (\sum a \bar{b} c \bar{d}) + (\sum a \bar{b} c d)\]

Two-level weakness

* Two-level representation is exponential
* However, it's a simple concept
  * Is \(\sum x = \text{odd}\)?
* Problem with representation, not function

Two-level representations also have other weaknesses
* Conversion from SOP to POS is difficult
  * Inverting functions is difficult
  * -ing two SOPs or +ing two POSs is difficult
* Neither general POS or SOP are canonical
  * Equivalence checking difficult
* POS satisfiability \(\in \mathcal{NP}\)-complete
Logic minimization motivation

- Want to reduce area, power consumption, delay of circuits
- Hard to exactly predict circuit area from equations
- Can approximate area with SOP cubes
- Minimize number of cubes and literals in each cube
- Algebraic simplification difficult
  - Hard to guarantee optimality

K-maps work well for small problems
- Too error-prone for large problems
- Don’t ensure optimal prime implicant selection
- Quine–McCluskey optimal and can be run by a computer
  - Too slow on large problems
- Espresso heuristic usually gets good results fast on large problems

Review: Algebraic simplification

Prove $XY + X\overline{Y} = X$

$XY + X\overline{Y} = X(Y + \overline{Y})$ distributive law
$X(Y + \overline{Y}) = X(1)$ complementary law
$X(1) = X$ identity law

Boolean function minimization

- Algebraic simplification
  - Not systematic
  - How do you know when optimal solution has been reached?
- Optimal algorithm, e.g., Quine–McCluskey
  - Only fast enough for small problems
  - Understanding these is foundation for understanding more advanced methods
- Not necessarily optimal heuristics
  - Fast enough to handle large problems

Quine–McCluskey two-level logic minimization

- Compute prime implicants with a well-defined algorithm
  - Start from minterms
  - Merge adjacent implicants until further merging impossible
- Select minimal cover from prime implicants
  - Unate covering problem

Definition: Unate covering

Given a matrix for which all entries are 0 or 1, find the minimum cardinality subset of columns such that, for every row, at least one column in the subset contains a 1.

I’ll give an example
### Cyclic core

```
  bc
0 0 0 1 11 10
0 0 0 1 11 10
1 1 1 0 0 0
1 1 1 0 0 0
```

### Implicant selection reduction
- Eliminate rows covered by essential columns
- Eliminate rows dominated by other rows
- Eliminate columns dominated by other columns

### Eliminate rows covered by essential columns

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>J</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>K</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

### Eliminate rows dominating other rows

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>J</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

### Eliminate columns dominated by other columns

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>J</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>K</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

### Backtracking
- Will proceed to complete solution unless cyclic
- If cyclic, can bound cover size
  - Compute independent sets

### Find lower bound

```
0X1 01X X01 1X0 1X0
001 1 1
011 1 1
010 1 1
100 1 1
101 1 1
110 1 1
```

3 disjoint rows → 3 columns required

### Use bound to constrain search space
- Eliminate rows covered by essential columns
- Eliminate rows dominated by other rows
- Eliminate columns dominated by other columns
- Branch-and-bound on cyclic problems
  - Use independent sets to bound
  - Speed improved, still $\in NP$-complete
Extremely brief introduction to \( \mathcal{NP} \)-completeness

- Any \( \mathcal{NP} \)-complete problem instance can be converted to any other \( \mathcal{NP} \)-complete problem instance in polynomial time (quickly)
- Nobody has ever developed a polynomial time (fast) algorithm that optimally solves an \( \mathcal{NP} \)-complete problem
- It is generally believed (but not proven) that it is not possible to devise a polynomial time (fast) algorithm that optimally solves an \( \mathcal{NP} \)-complete problem
- Can use heuristics
  - Fast algorithms that often produce good solutions

There also exist exponential-time algorithms: \( \mathcal{O}(2^n) \), \( \mathcal{O}(2^n \cdot \log n) \), \( \mathcal{O}(n^3) \)

\( \mathcal{NP} \)-completeness

Recall that sorting may be done in \( \mathcal{O}(n \log n) \) time
DFS \( \in \mathcal{O}(|V| + |E|) \), BFS \( \in \mathcal{O}(|V|) \), Topological sort \( \in \mathcal{O}(|V| + |E|) \)

For \( t(n) = 2^n \) seconds
\[ t(1) = 2 \text{ seconds} \]
\[ t(10) = 17 \text{ minutes} \]
\[ t(20) = 12 \text{ days} \]
\[ t(50) = 35,702,052 \text{ years} \]
\[ t(100) = 40,196,936,841,331,500,000,000 \text{ years} \]

\( \mathcal{NP} \)-completeness

- Digital design and synthesis is full of \( \mathcal{NP} \)-complete problems
- Graph coloring
- Scheduling
- Graph partitioning
- Satisfiability (and 3SAT)
- Covering
- \ldots and many more

- There is a class of problems, \( \mathcal{NP} \)-complete, for which nobody has found polynomial time solutions
- It is possible to convert between these problems in polynomial time
- Thus, if it is possible to solve any problem in \( \mathcal{NP} \)-complete in polynomial time, all can be solved in polynomial time
- Unproven conjecture: \( \mathcal{NP} \neq \mathcal{P} \)
• What is \( \mathcal{NP} \)? Nondeterministic polynomial time.

• A computer that can simultaneously follow multiple paths in a solution space exploration tree is nondeterministic. Such a computer can solve \( \mathcal{NP} \) problems in polynomial time.

• I.e., a computer that can simultaneously be in multiple states.

• Nobody has been able to prove either \( P \neq \mathcal{NP} \) or \( P = \mathcal{NP} \).

If we define \( \mathcal{NP} \)-complete to be a set of problems in \( \mathcal{NP} \) for which any problem’s instance may be converted to an instance of another problem in \( \mathcal{NP} \)-complete in polynomial time, then

\[
P \subseteq \mathcal{NP} \rightarrow \mathcal{NP} \text{-complete} \cap P = \emptyset
\]

Basic complexity classes

• \( P \) solvable in polynomial time by a computer (Turing Machine)

• \( \mathcal{NP} \) solvable in polynomial time by a nondeterministic computer

• \( \mathcal{NP} \)-complete converted to other \( \mathcal{NP} \)-complete problems in polynomial time

How to deal with hard problems

• What should you do when you encounter an apparently hard problem?

• Is it in \( \mathcal{NP} \)-complete?

• If not, solve it

• If so, then what?

Despair. Solve it! Resort to a suboptimal heuristic. Bad, but sometimes the only choice. Develop an approximation algorithm.

Better. Determine whether all encountered problem instances are constrained. Wonderful when it works.

One example


Optimal two-level logic synthesis is \( \mathcal{NP} \)-complete

• Upper bound on number of prime implicants grows \( \frac{3^n}{n} \) where \( n \) is the number of inputs

• Given > 16 inputs, can be intractable

• However, there have been advances in complete solvers for many functions

  • Optimal solutions are possible for some large functions

Heuristic logic minimization

• For difficult and large functions, solve by heuristic search

• Multi-level logic minimization is also best solved by search

  The general search problem can be introduced via two-level minimization

  • Examine simplified version of the algorithms in Espresso

  • Generate only a subset of prime implicants

  • Carefully selects subset of prime implicants covering on-set

  • Guaranteed to be correct

  • May not be optimal
Can be viewed in the following:
- Start with a potentially optimal algorithm
- Add numerous techniques for constraining the search space
- Uses efficient move order to allow pruning
- Disable backtracking to arrive at a heuristic solver
- Widely used in industry
- Still has room for improvement
  - E.g., early recursion termination

Properties of two-level logic
- The Quine-McCluskey (tabular) method
- \( \mathcal{NP} \)-complete: Why use heuristics?
- Espresso

Homework assignment one
- Algebraic manipulation (Review)
- K-Maps (Review)
- Quine-McCluskey
- Espresso

Next lecture
- More on Espresso algorithm
- Technologies and implementation methods

Reading assignment
- More Optimization for Quine–McCluskey