Logic Simplification

Logic Expression **Simplification**

- Often it is possible to simplify function expressions from, say, their minterm form.
- This may be done by using various identities, as we know. But more systematic methods will be shown.

Simplification Example

3-argument majority function

v	w	x	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

minterm form: majority(v, w, x) =

Notice Certain “Adjacencies”

3-argument
majority function

v	w	x	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

These rows are “adjacent”: their variable combinations differ in only 1 bit

v’wx

vwx

= wx

simplified majority(v, w, x) = $\boxed{\text{wx}}$ + vw’x + vwx’

unsimplified majority(v, w, x) = $\boxed{\text{v'wx}}$ + vw’x + vwx’ + $\boxed{\text{vwx}}$

Any other Adjacencies?

(ok to use a row more than once)

v	w	x	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Adjacencies can Overlap

3-argument
majority function

v	w	x	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

simplified majority(v, w, x) = $wx + \boxed{vx} + vwx'$

unsimplified majority(v, w, x) = $v'wx + \boxed{vw'x} + vwx' + \boxed{vwx}$

Any More Adjacencies?

3-argument
majority function

v	w	x	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

simplified majority(v, w, x) = wx + $\boxed{\text{vx}}$ + vwx'

unsimplified majority(v, w, x) = v'wx + $\boxed{\text{vw'x}}$ + vwx' + $\boxed{\text{vwx}}$

Any More Adjacencies?

3-argument
majority function

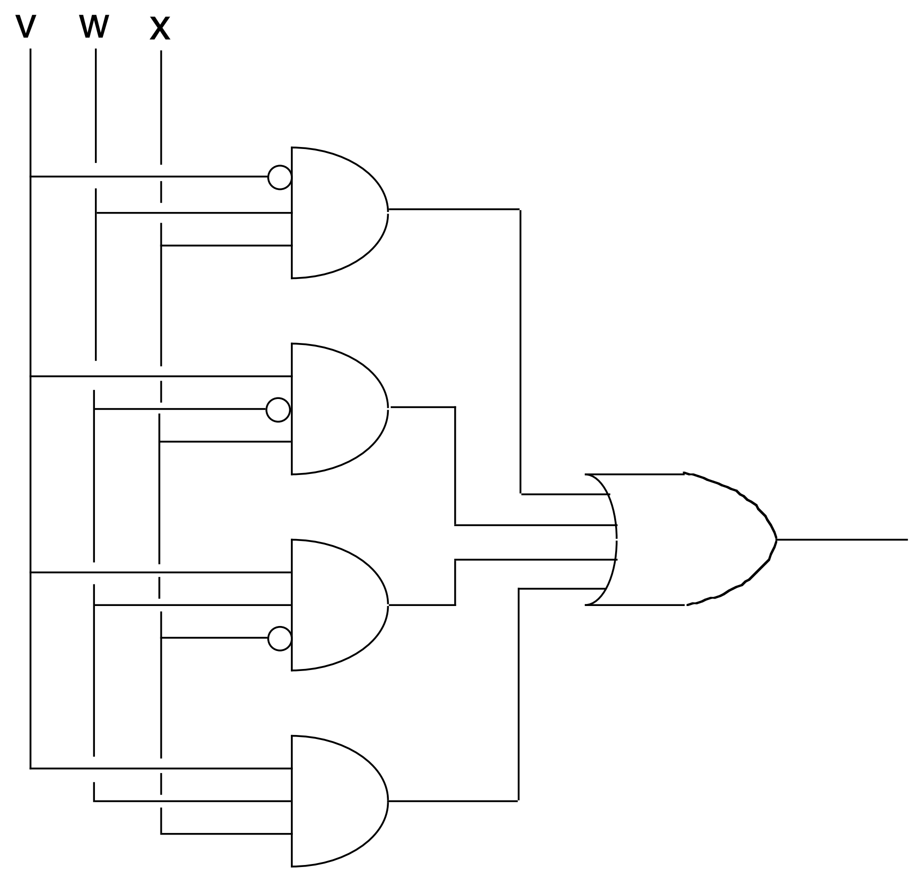
v	w	x	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

simplified majority(v, w, x) = wx + vx + $\boxed{\text{vw}}$

unsimplified majority(v, w, x) = v'wx + vw'x + $\boxed{\text{vwx'}}$ + $\boxed{\text{vwx}}$

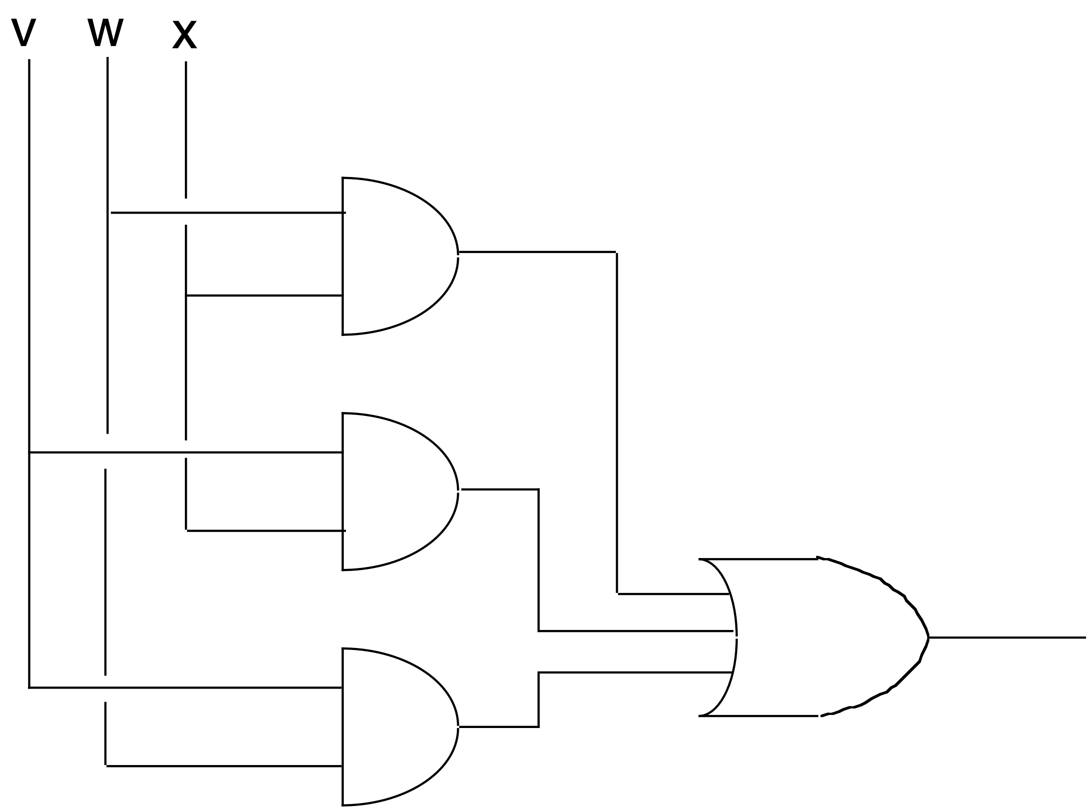
unsimplified majority

majority(v, w, x) = v'wx + vw'x + vwx' + vwx



simplified majority

majority(v, w, x) = wx + vx + vw



Simplified/Unsimplified Comparison for Majority

- Draw the diagrams
- Unsimplified:
  - 4 and-gates, 3 inputs each,
    - some with negation,
  - 1 4-input or-gate
- Simplified:
  - 3 and-gates, 2 inputs each,
    - none with negation,
  - 1 3-input or-gate

Implications for Simplified Functions

- If communicated by human, easier to understand, transfer
- If implemented as hardware, fewer gates, wires
- If implemented as software, fewer tests, execution steps

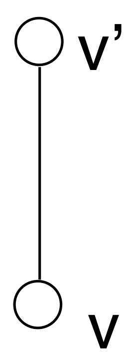
Visualizing Adjacencies

- Hypercube representation of truth table
- Karnaugh map representation

Hypercubes
(connected vertices = adjacent)

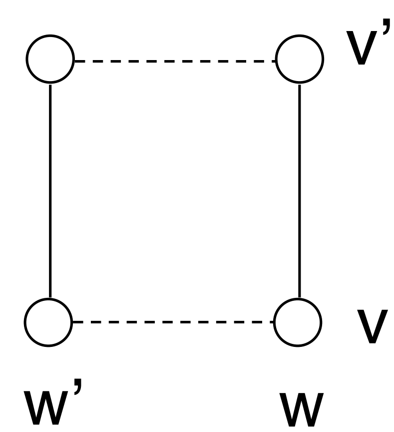
1-dimension =

1 logical variable



2-dimensions =

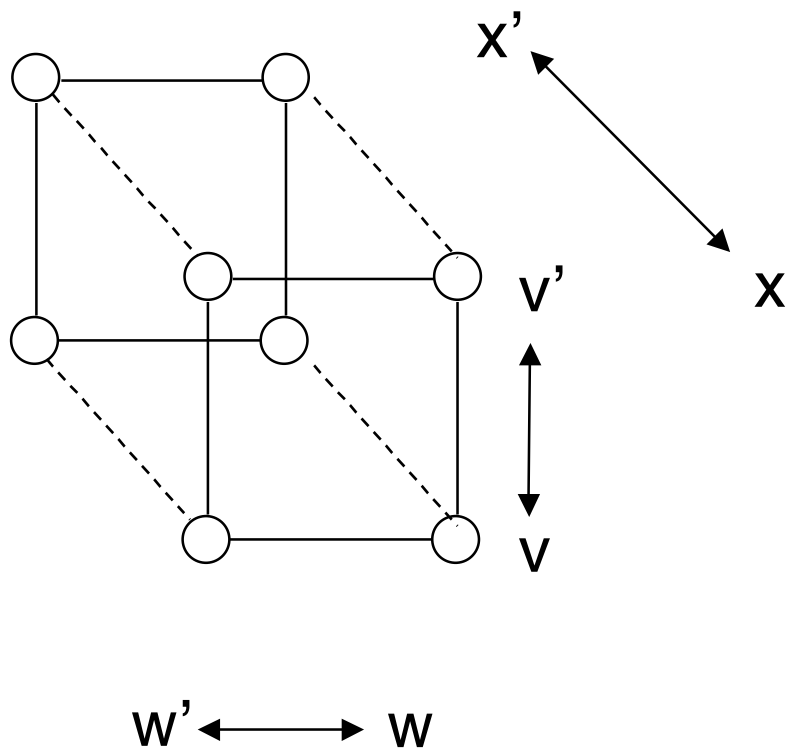
2 logical variables



Hypercubes

3-dimensions =

3 logical variables

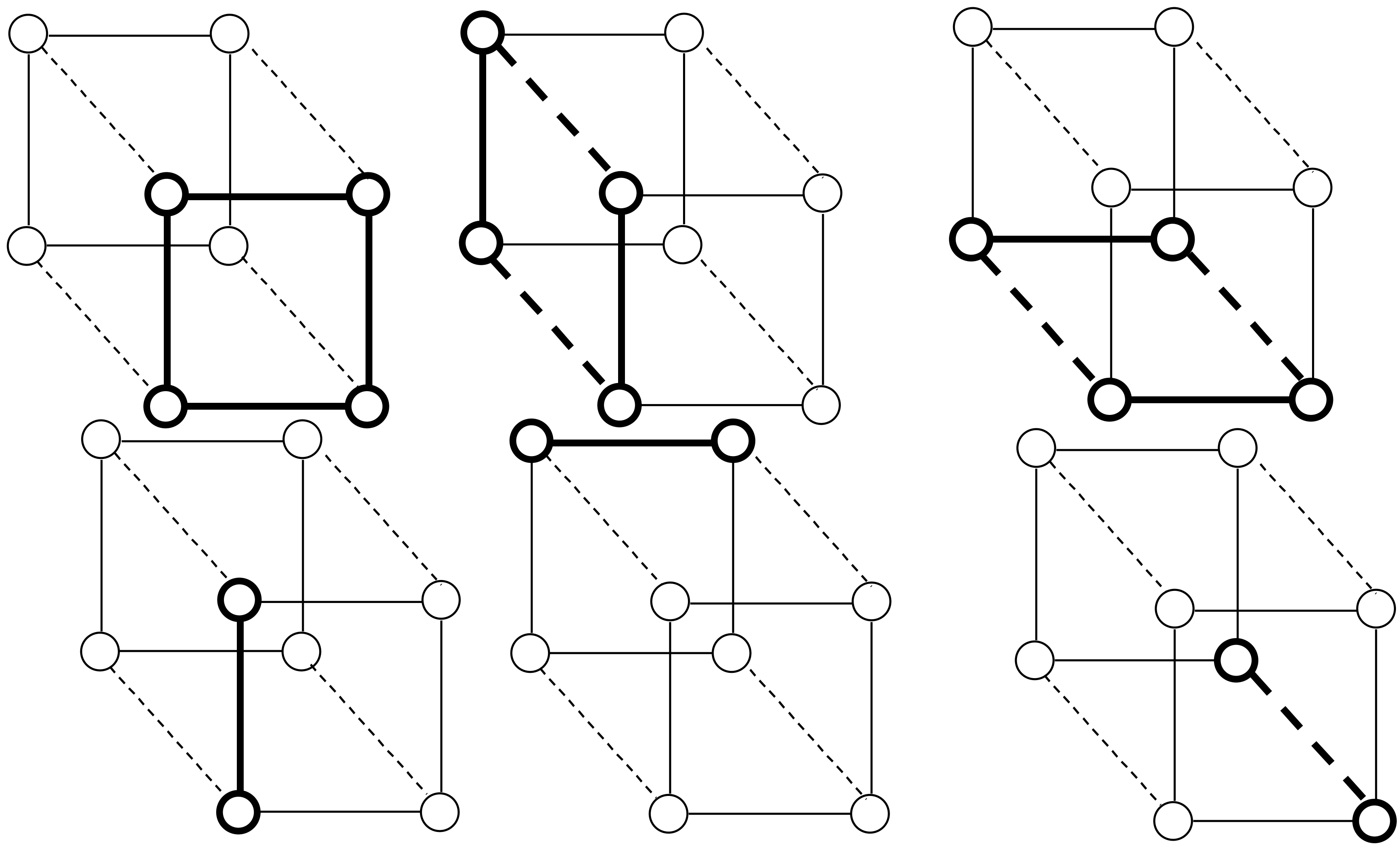


Sub-cubes

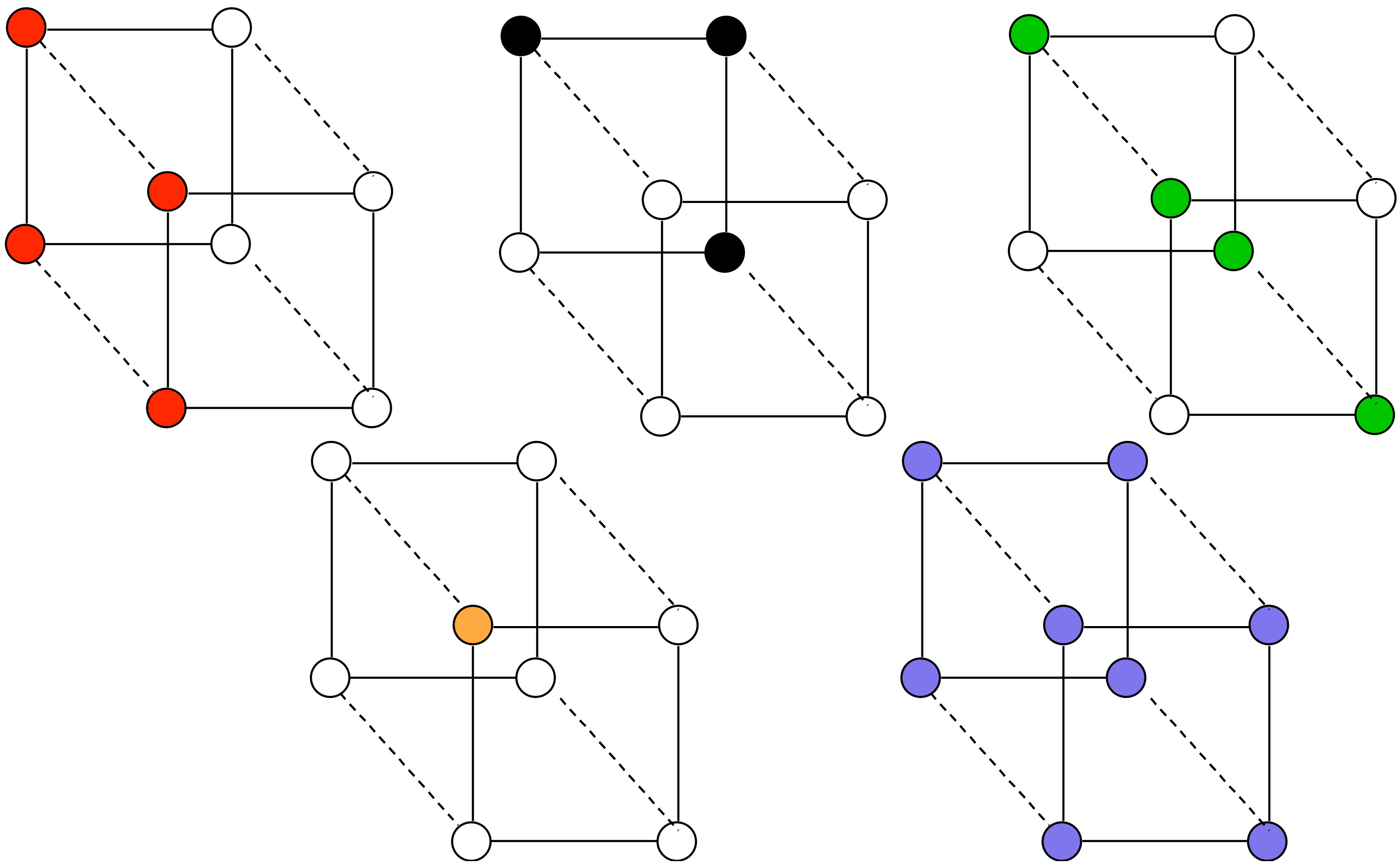
- An n-dimensional hypercube, for $n > 0$, has embedded in it a set of m-dimensional hypercubes, for each $m < n$.

- These are call the ***sub-cubes*** of the original hypercube.

Sub-cubes



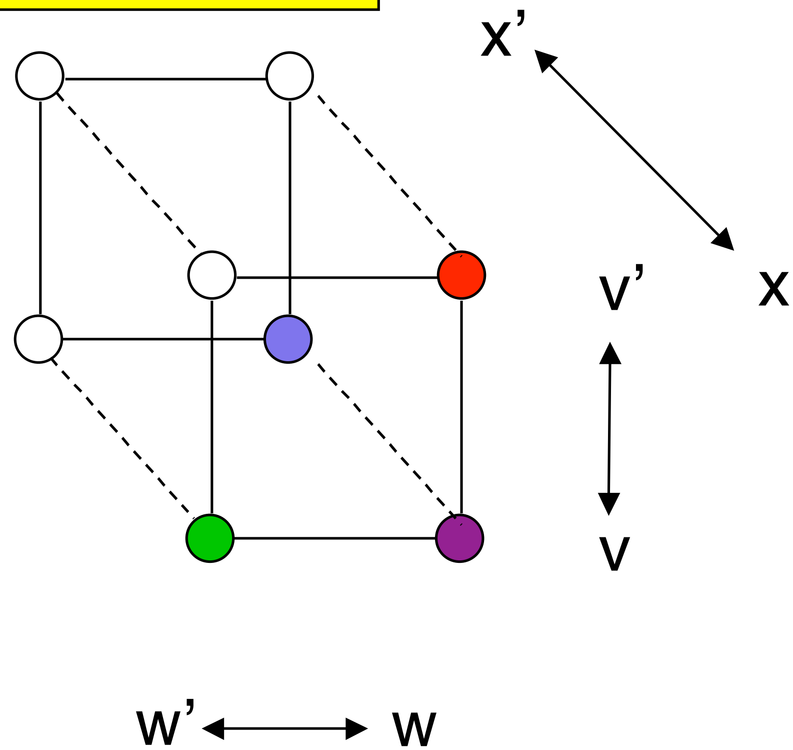
Which sets are subcubes?



"Plotting" Functions on Hypercubes

majority(v, w, x) = v'wx + vw'x + vwx' + vwx

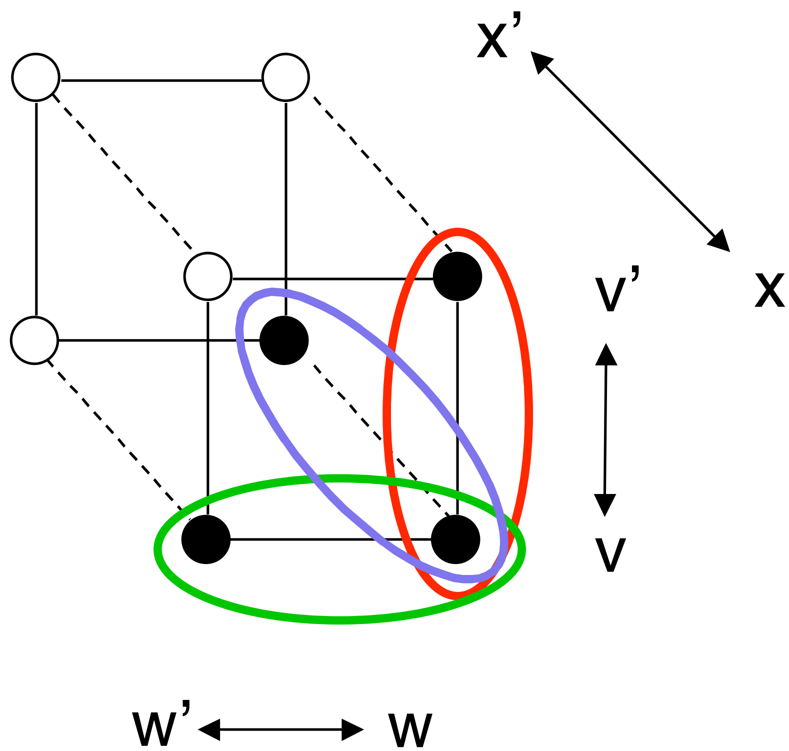
each minterm is a vertex



Simplified Functions on Hypercubes

majority(v, w, x) = wx + vx + vw

each term is a ***sub-cube***



Sub-Cube Dimensions

- The **dimension of a sub-cube** is

  n minus (number of variables in the corresponding product term)

- where n is the total number of variables.
- Examples: 3 variables total:
  - dimension 0: 3 variables
  - dimension 1: 2 variables
  - dimension 2: 1 variables
  - dimension 3: 0 variables (= constant 1)

- **“More is less”** (Greater dimension, fewer variables)

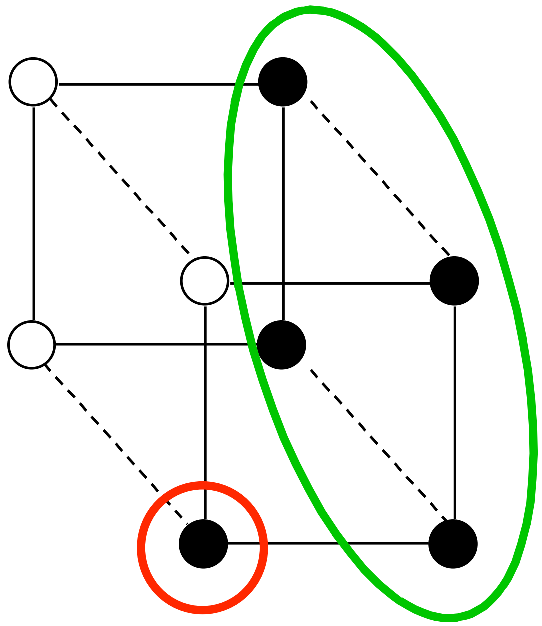
SOP Circuits
(sum-of-products)

- An SOP will always correspond to
  - a set of **sub-cubes**
  - Collectively the vertices in the sub-cubes *cover* the "on" vertices.
  - The vertices in the sub-cubes don't cover any "off" vertices.
- Such a set is called a **cover** for the function.
- Each cover corresponds to a different gate implementation.

Simplified Covers

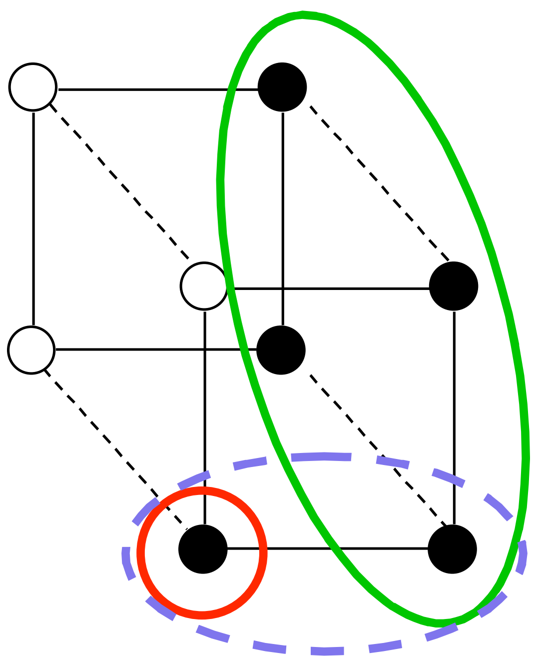
- In a fully-simplified SOP, each sub-cube will be ***maximal***, in that it is not contained within another sub-cube.
- If instead it were properly contained in another sub-cube, then it could be **replaced** with that sub-cube.
- The replacement would have **fewer** variables, i.e. be simpler.

Maximality



This set of 2 sub-cubes covers the function.

Maximality

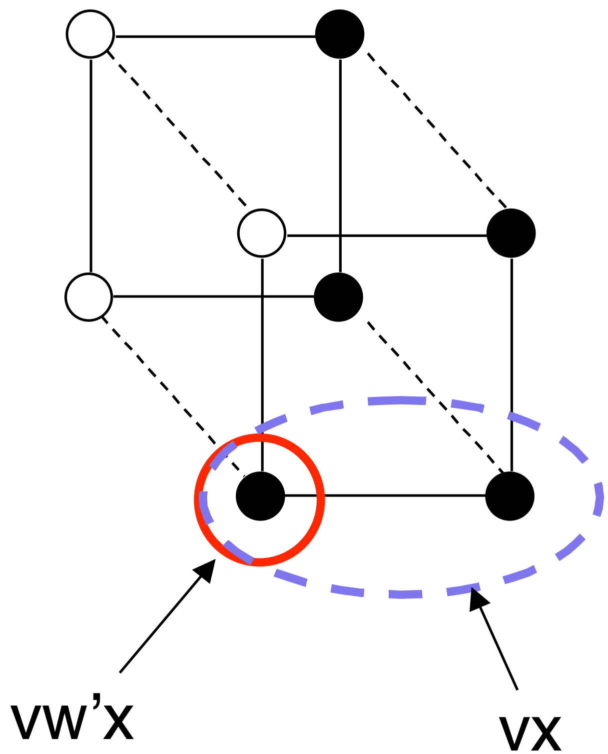


However the red sub-cube is not maximal.

It should be extended to the blue sub-cube as shown for greater simplification.

The resulting cover is simpler, even though one vertex is covered twice.

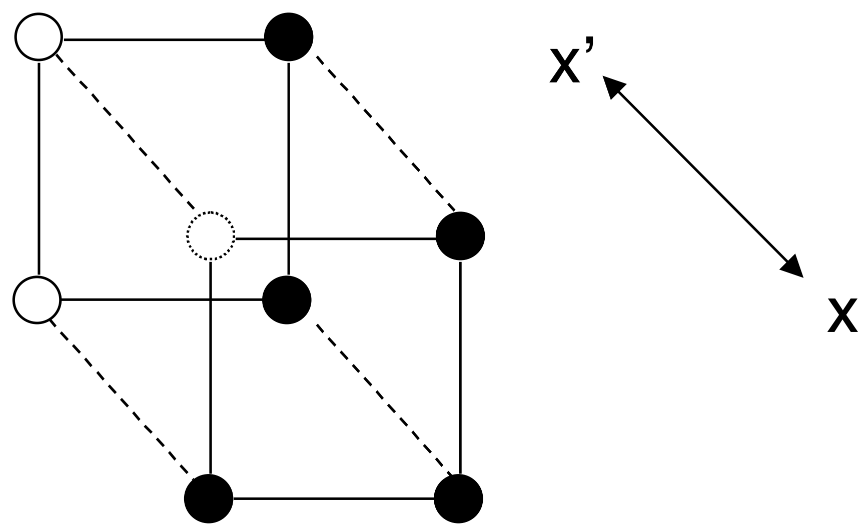
More-is-Less Principle



Making the sub-cubes as **large** as possible, makes the resulting product term as **small** as possible.

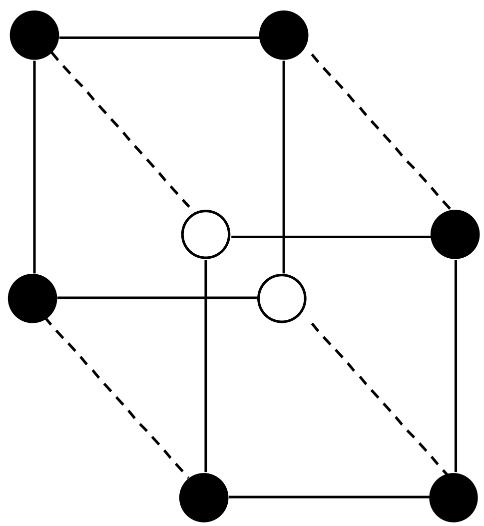
Einstein's Principle:

Explanations should be made as simple as possible, but no simpler.

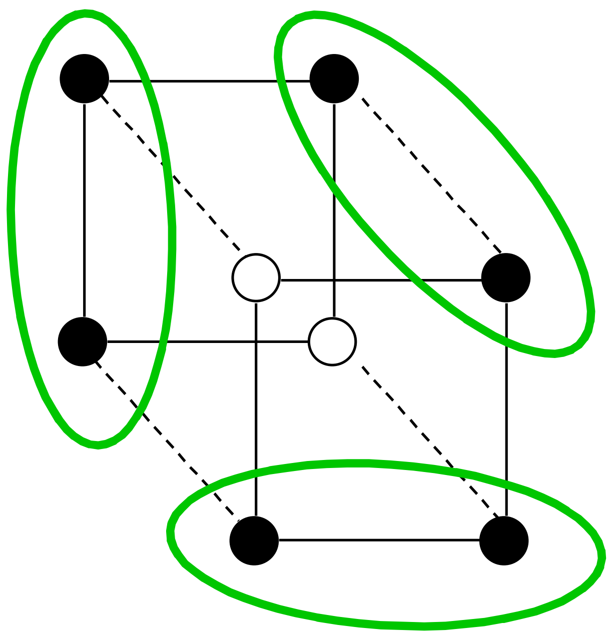


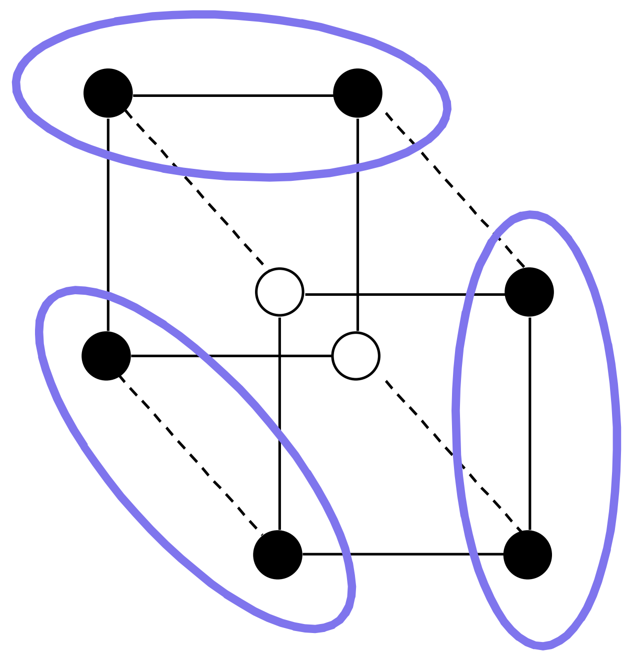
Including x by itself will not work!

Non-Uniqueness of Simplest Cover

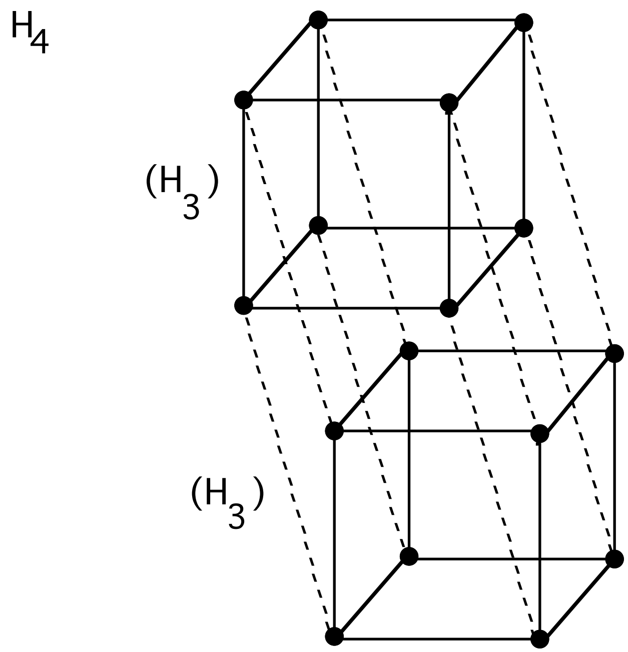


Non-Uniqueness of Simplest Cover

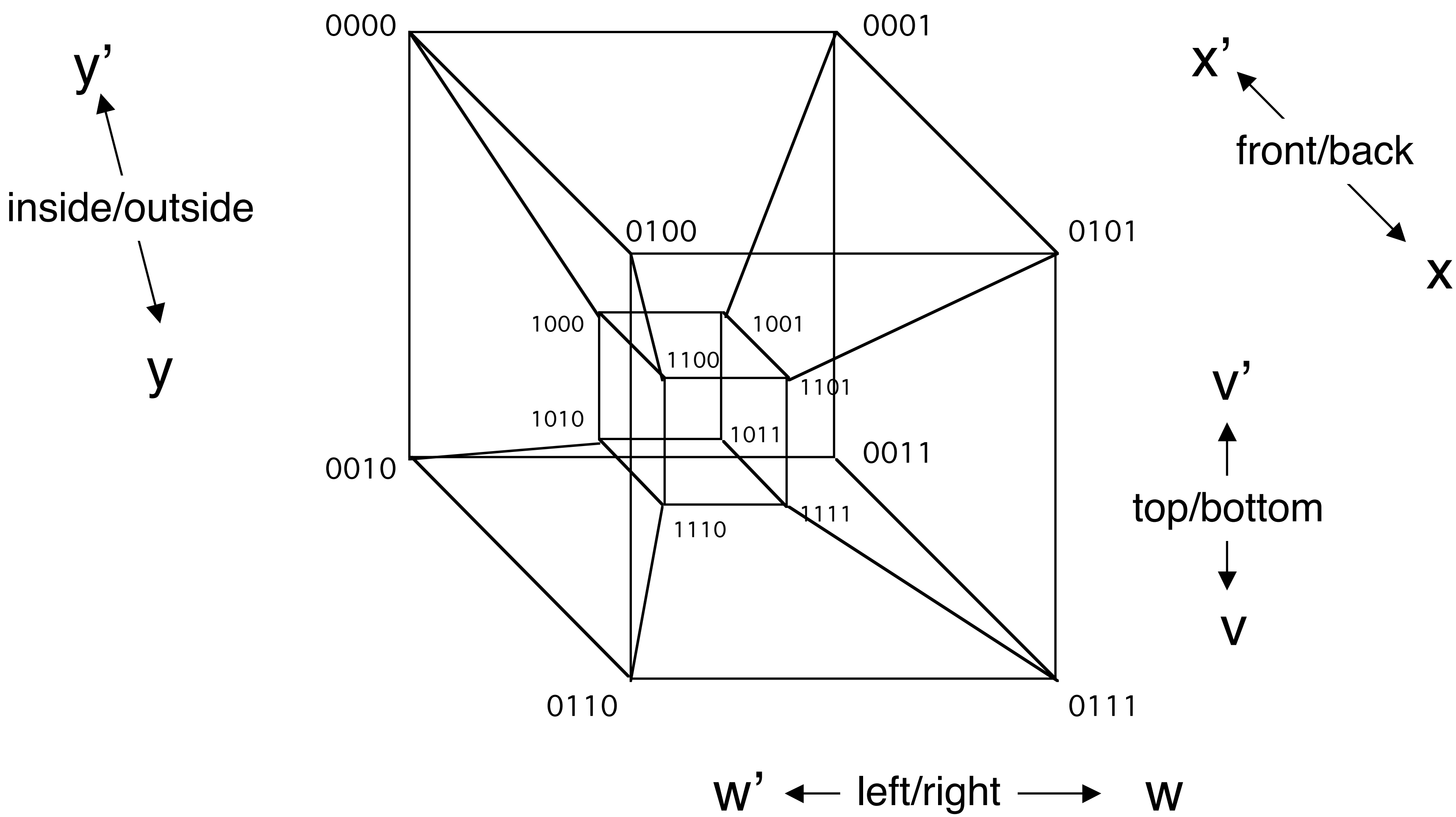




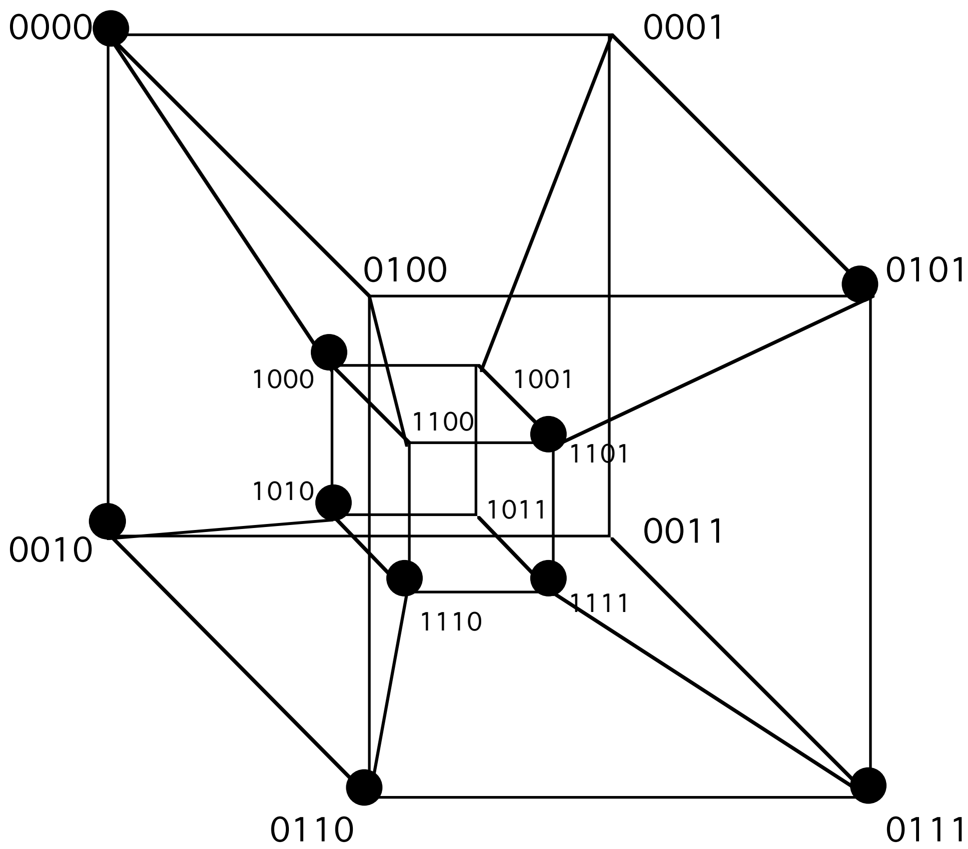
4-D Hypercube



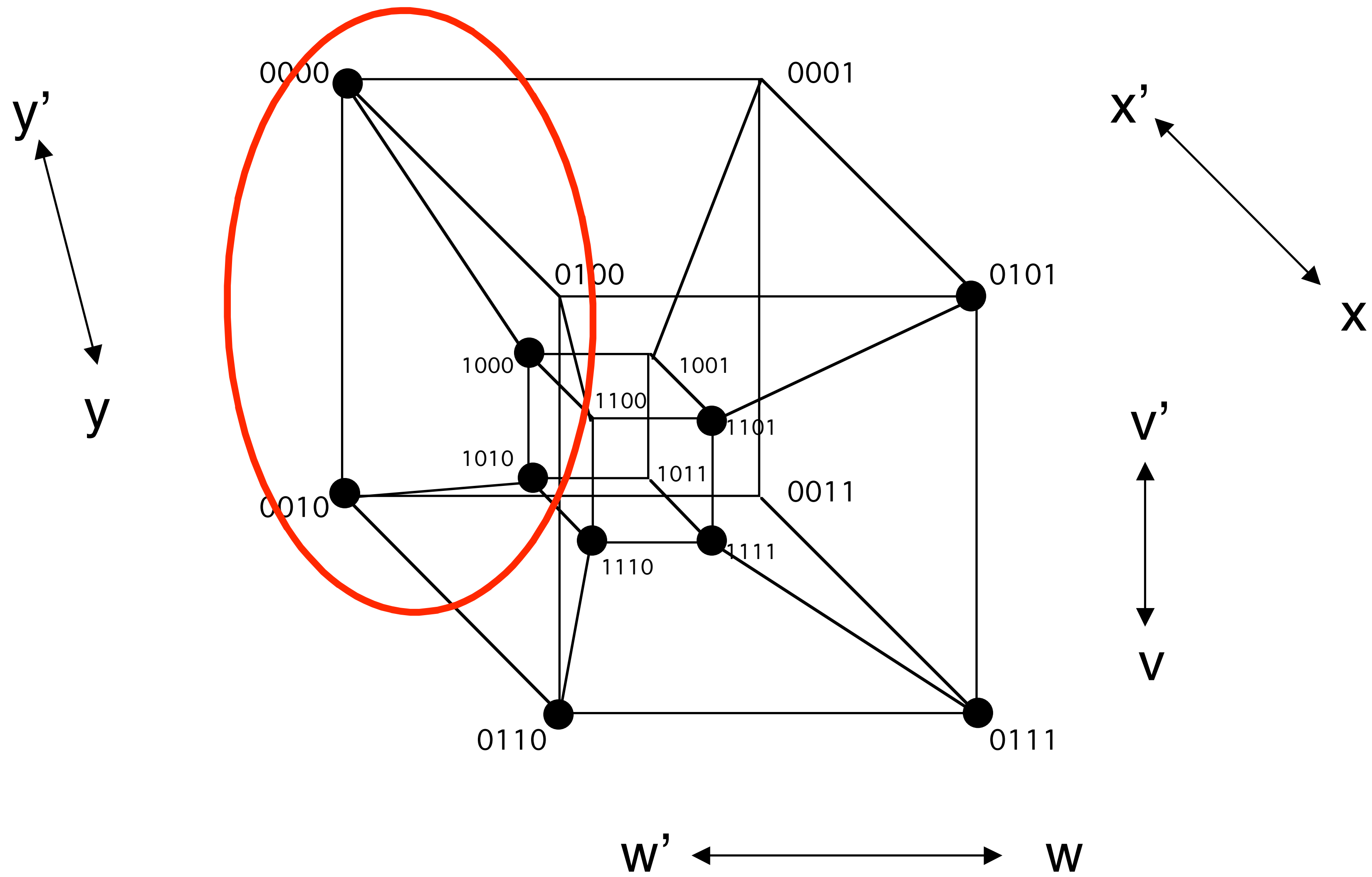
Alternate 4-D hypercube representation
(= "shadow" of a Tesseract)



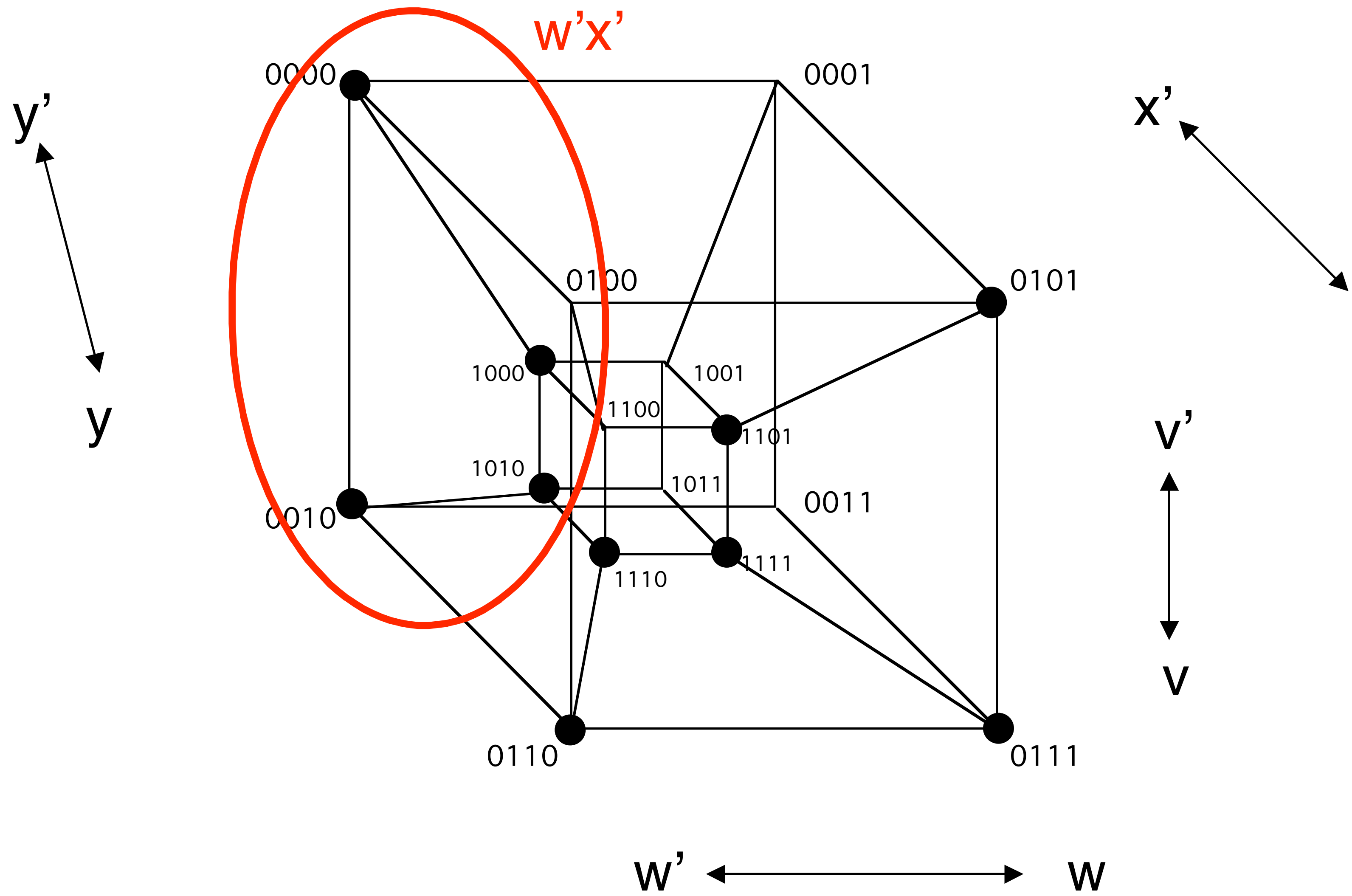
Plotting Function on 4-D Hypercube



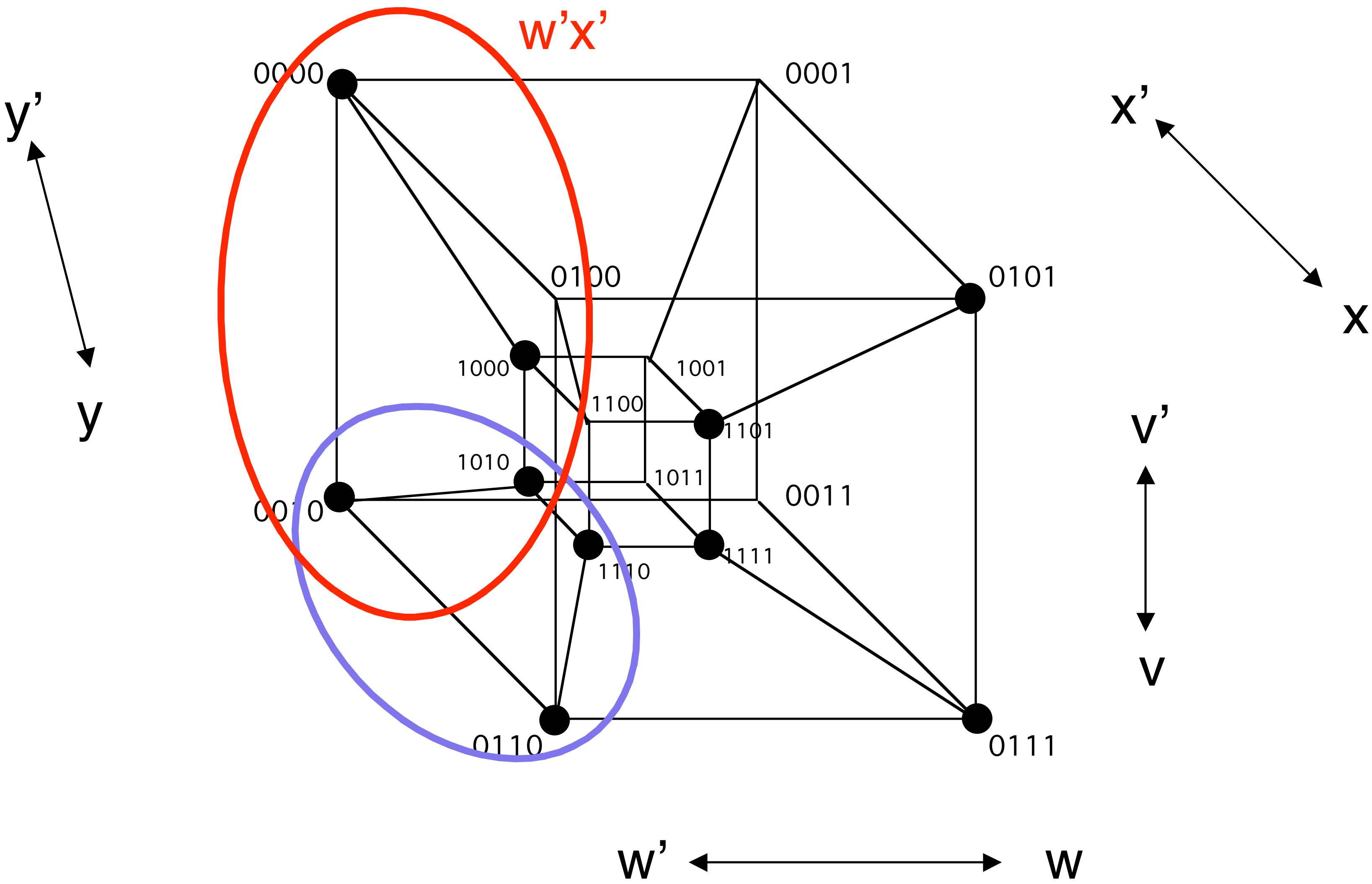
Plotting Function on 4-D Hypercube



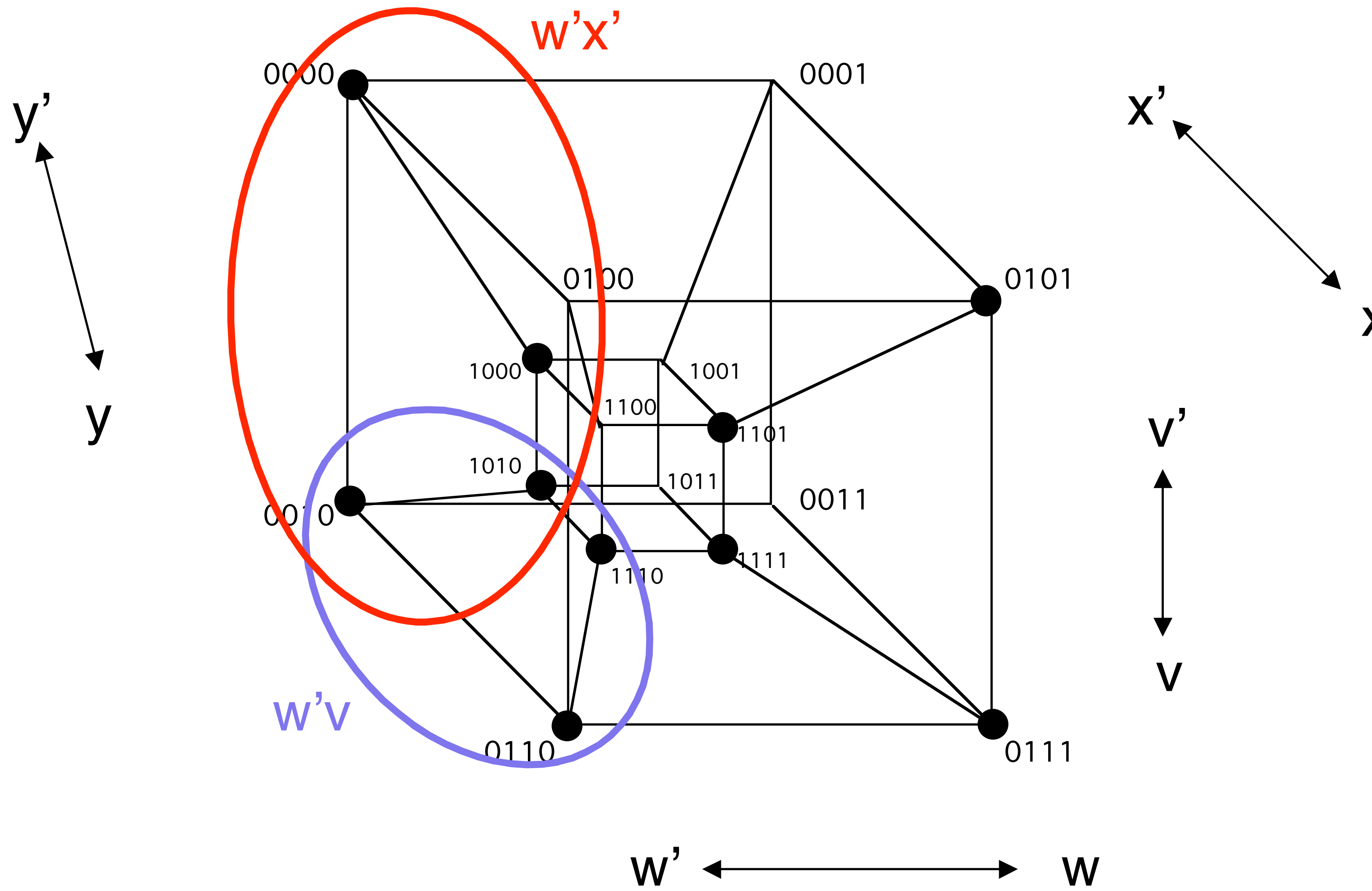
Plotting Function on 4-D Hypercube



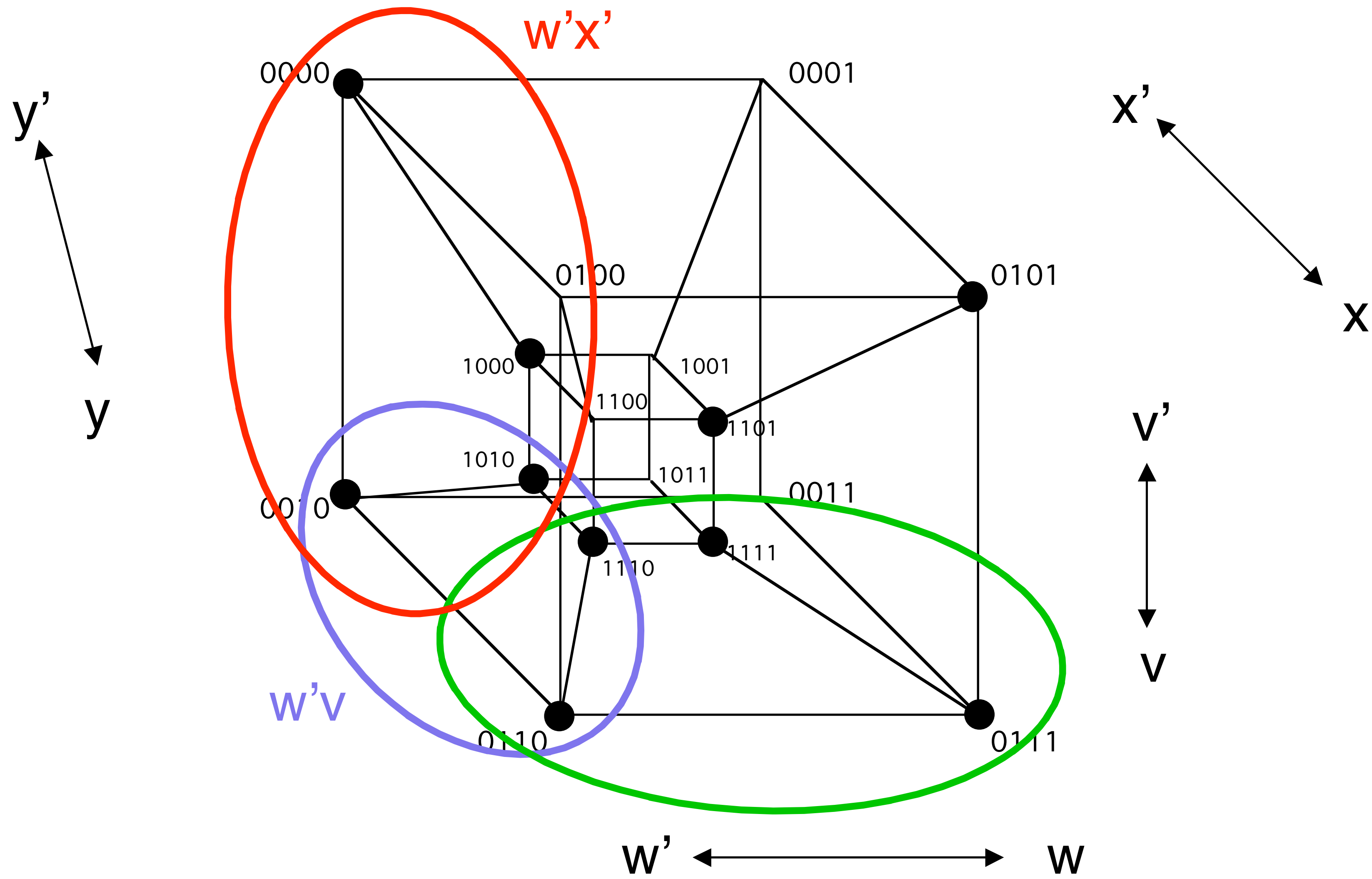
Plotting Function on 4-D Hypercube



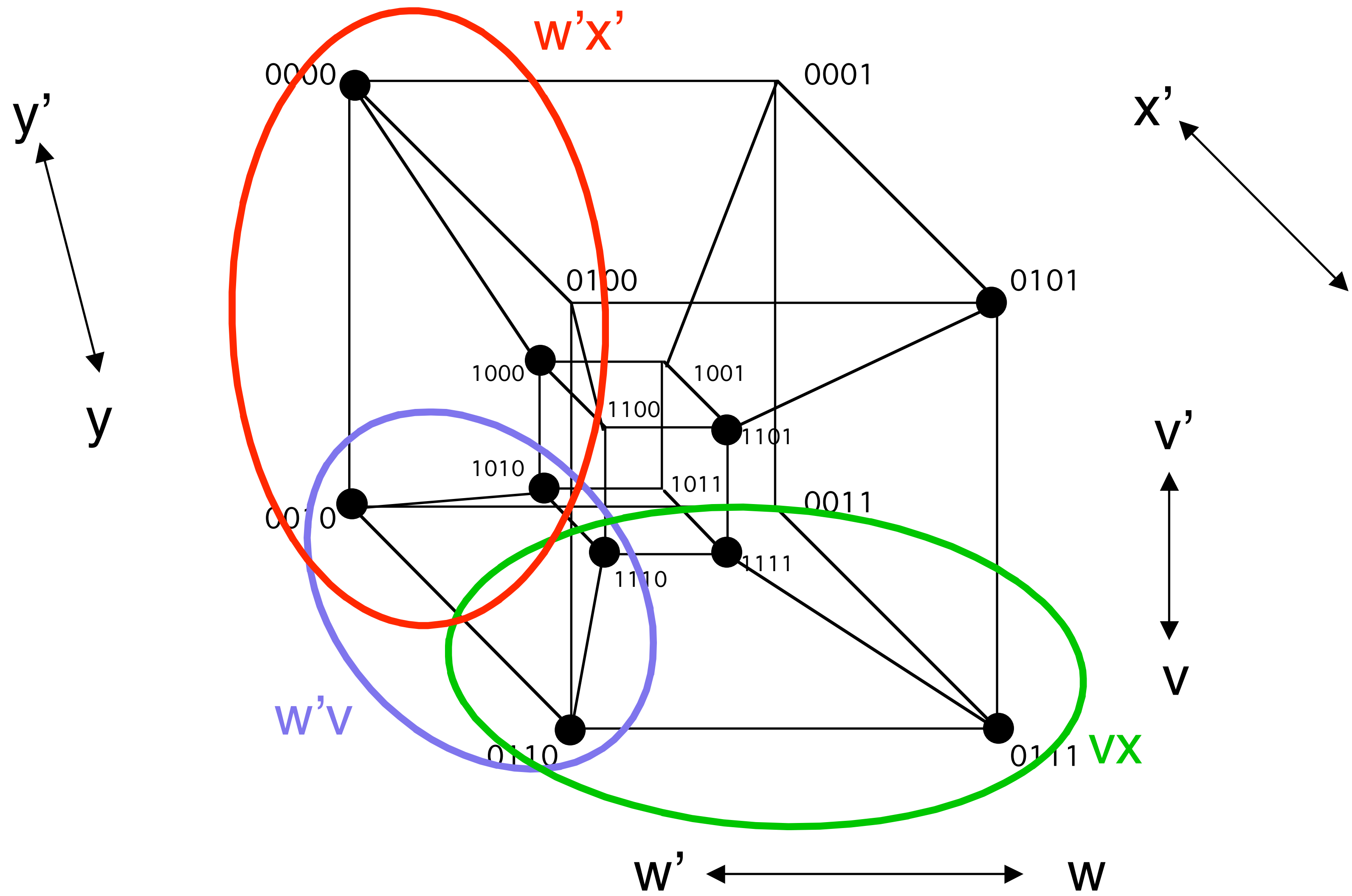
Plotting Function on 4-D Hypercube



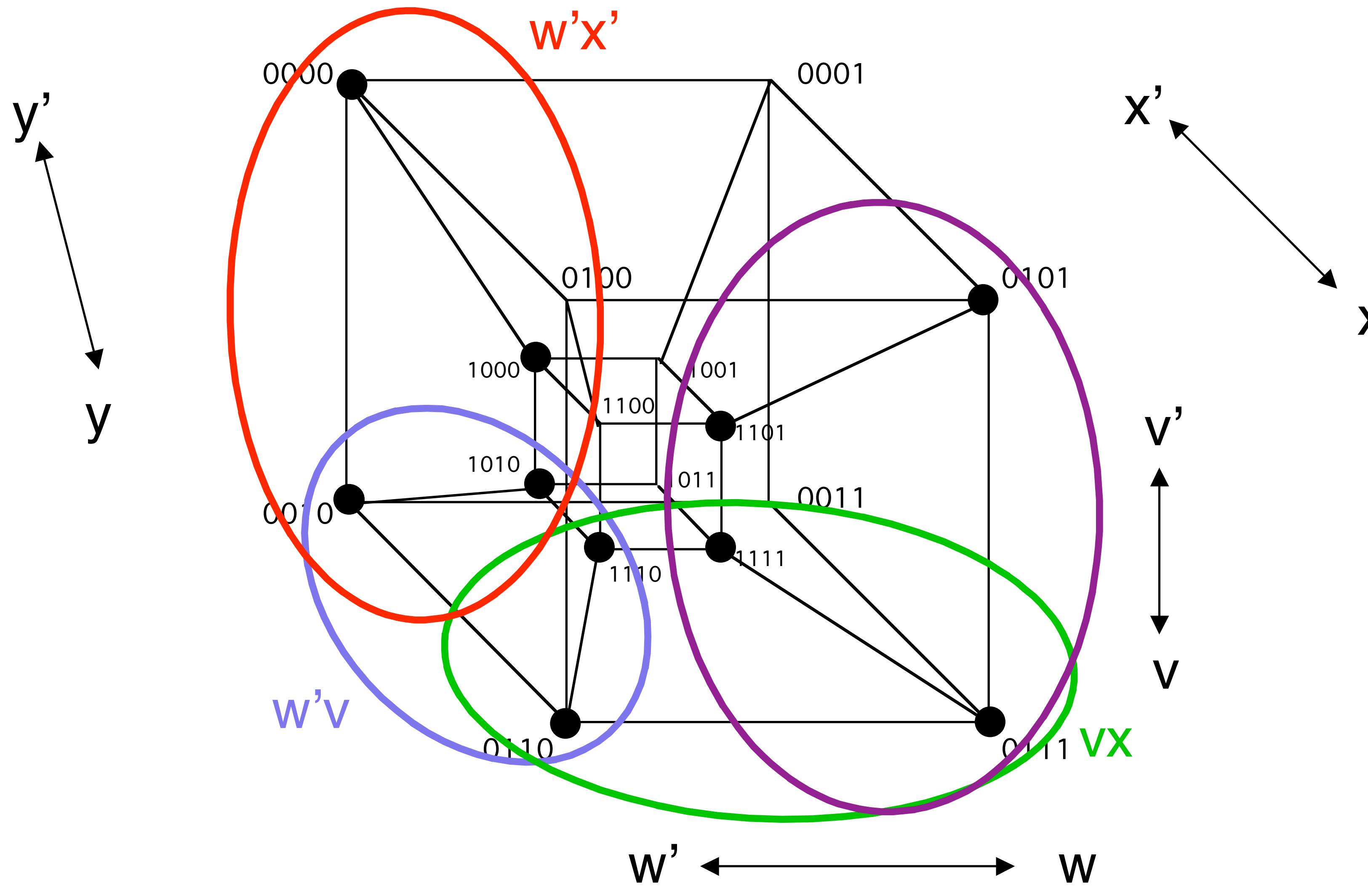
Plotting Function on 4-D Hypercube



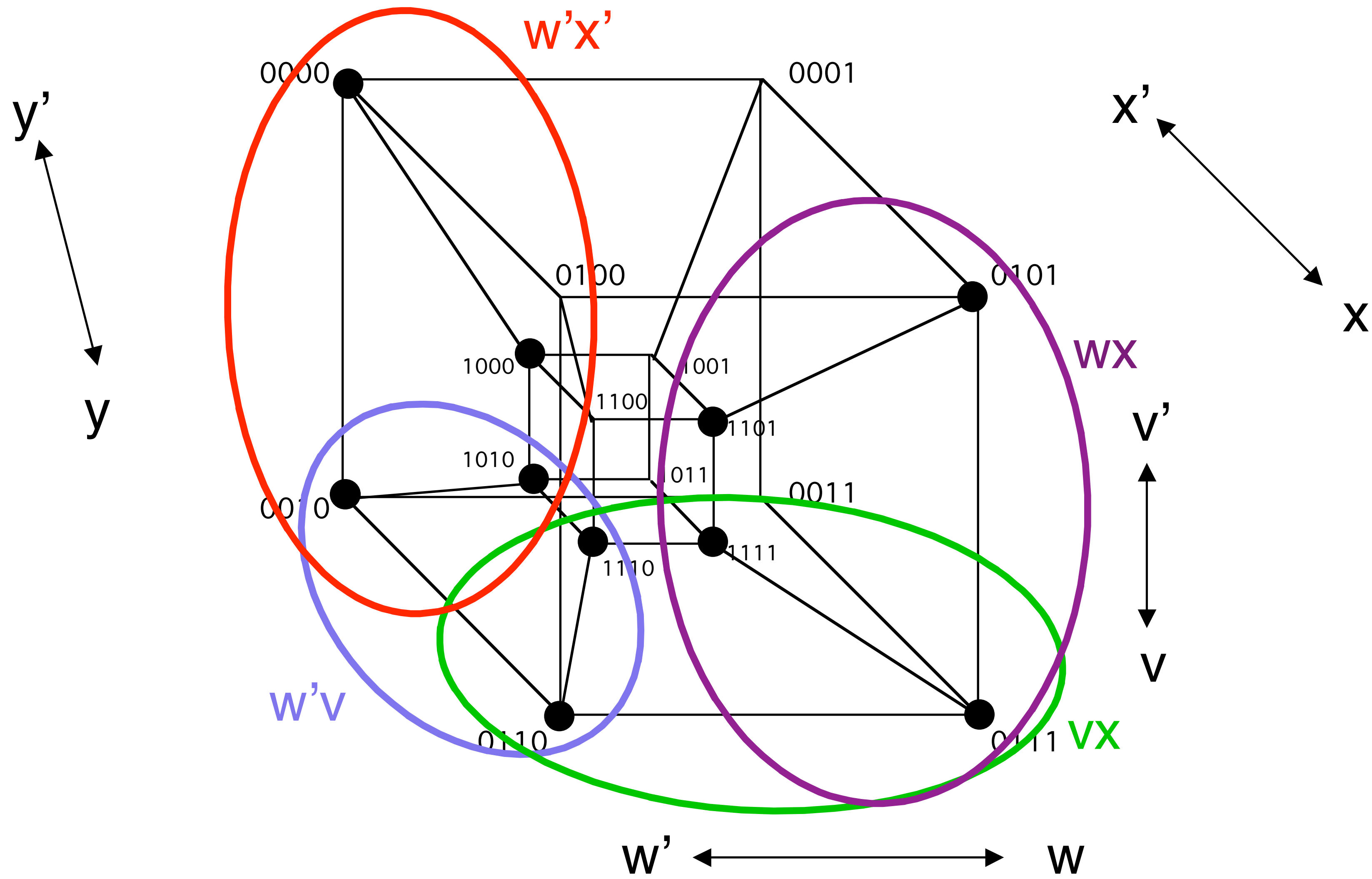
Plotting Function on 4-D Hypercube



Plotting Function on 4-D Hypercube



Plotting Function on 4-D Hypercube



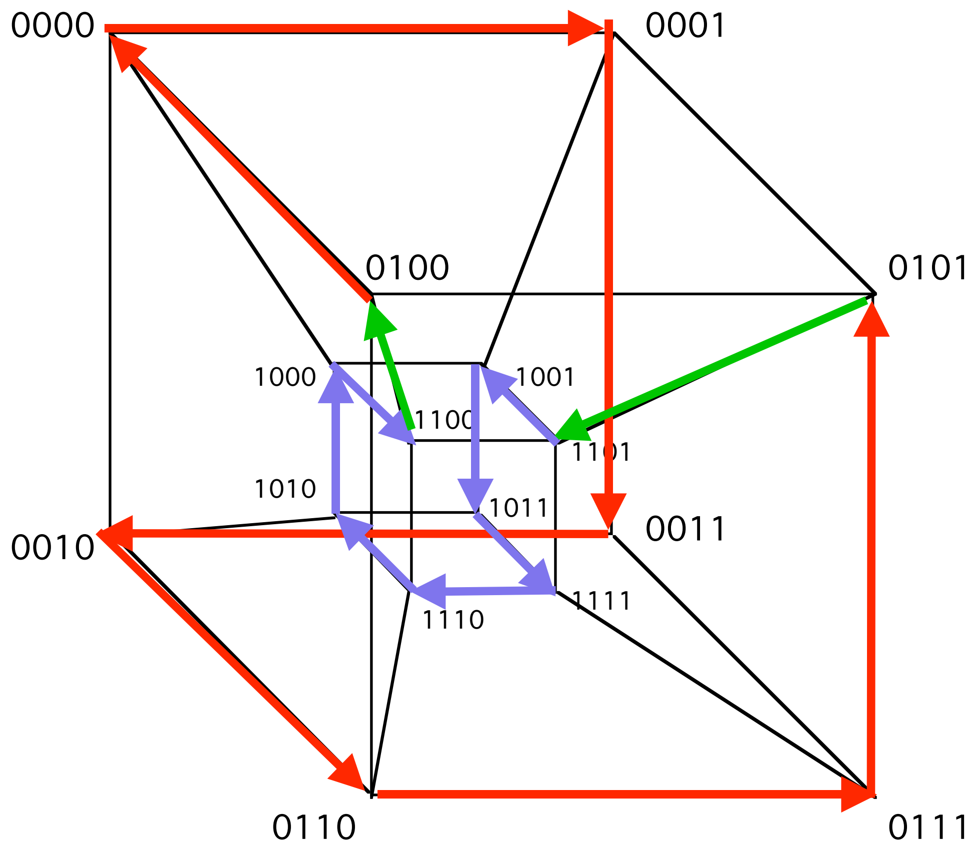
Hypercube Function-Plotting Applet

An applet conceived by R. Keller and implemented by Ian Weiner, HMC '01 as a CS60 project.

http://www.cs.hmc.edu/~keller/javaExamples/hypercube

Check out dimensions > 4.

Gray Code on a Hypercube

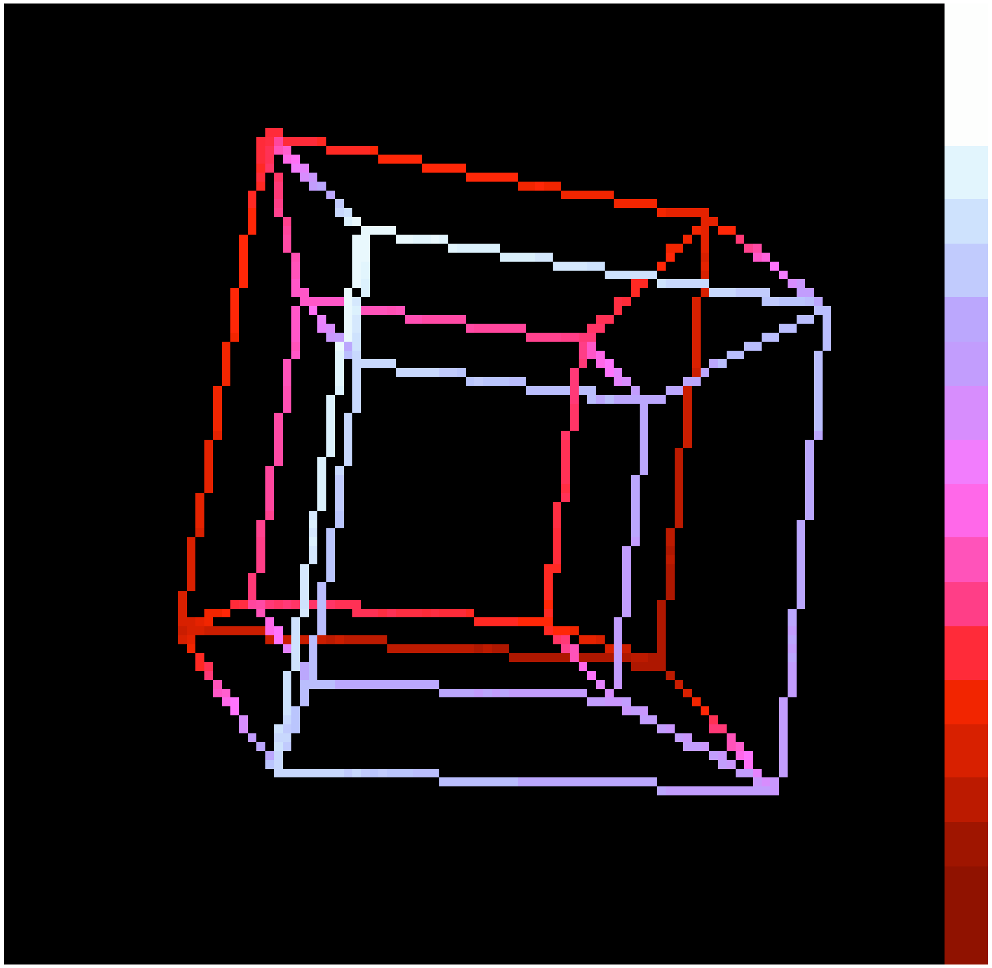


Path touches every node without retracing any arc (Hamiltonian path).

Other CS uses of Hypercubes

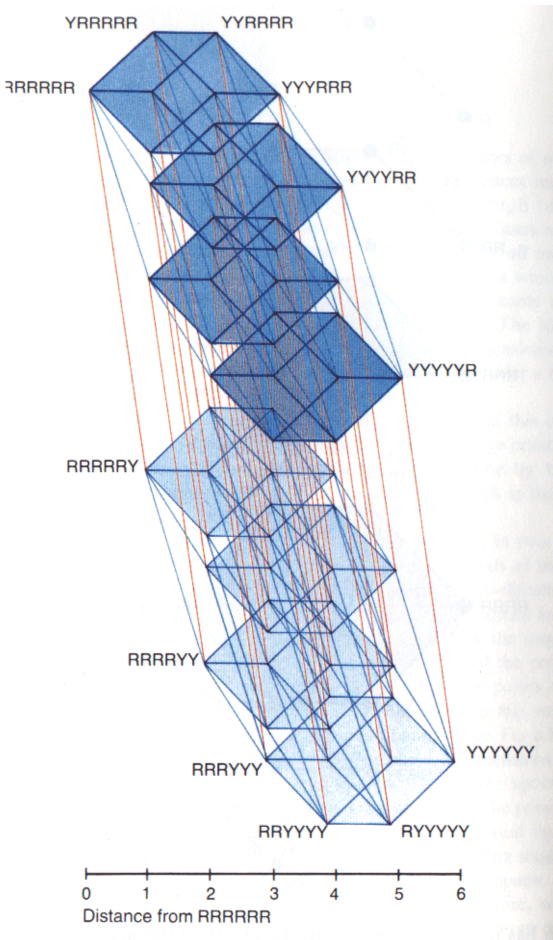
- Distributed parallel computer topologies
- High-speed sorting

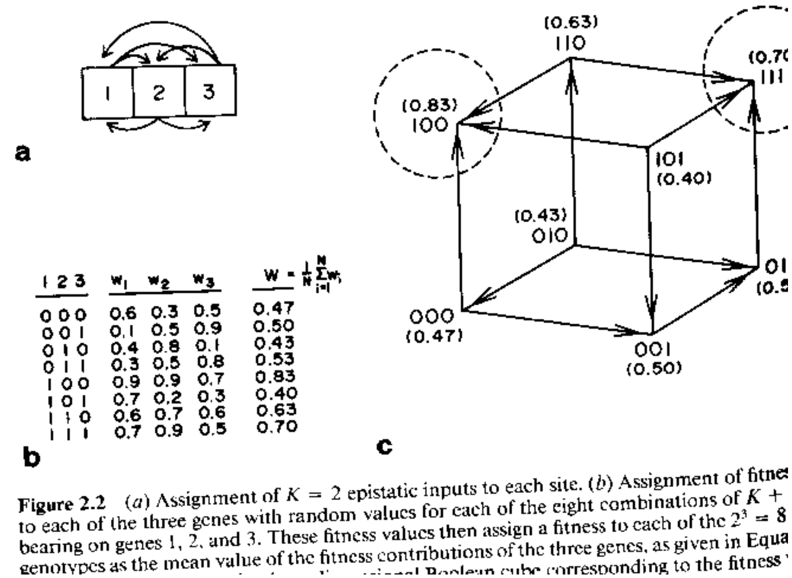
Non-CS uses of Hypercubes



Rotating hypercube from http://casa.colorado.edu/~ajsh/sr/hypercubel.html

Hypercubes occur wherever independent attributes are found: e.g. genetics, “complexity”



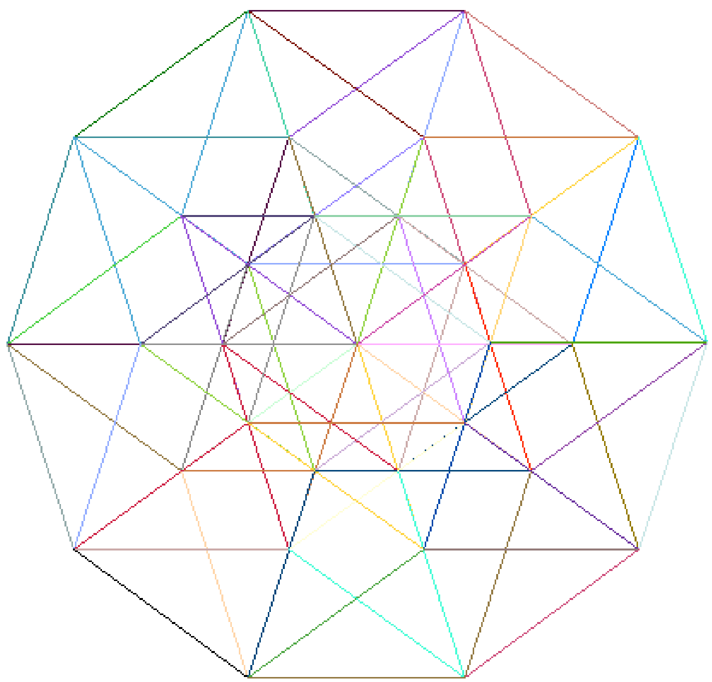


Left: A 6-dimensional hypercube representing a genetic *sequence space*. From Manfred Eigen, *Steps toward life*, Oxford U. Press, 1992, p. 94.

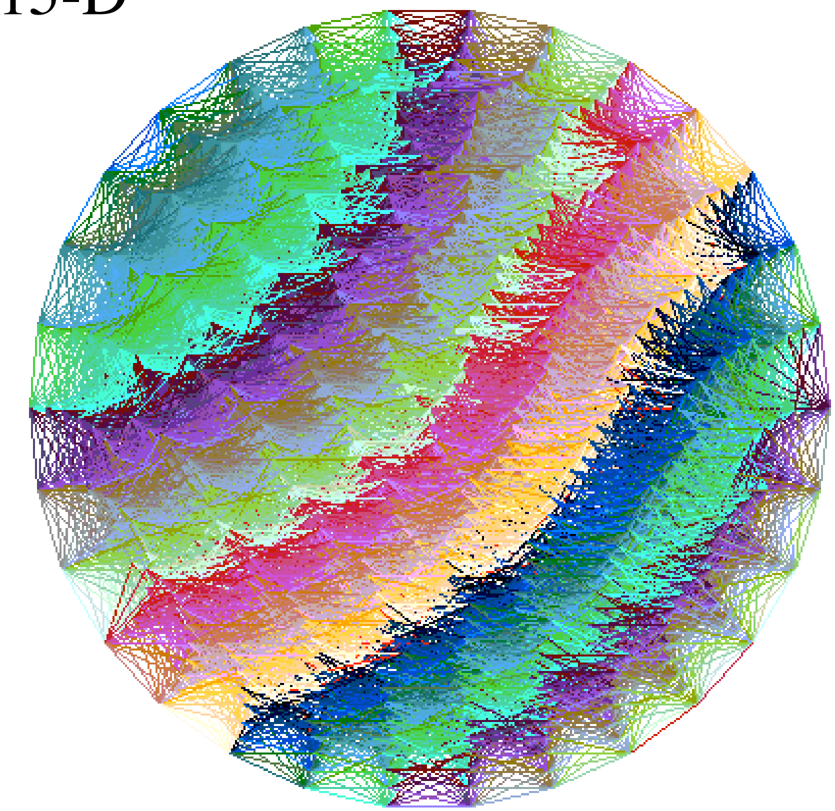
Above: Hypercube representation of fitness landscape of mutants. From Stuart A. Kauffman, *The origins of order*, Oxford U. Press, 1993, p. 42.

More Hypercube Projections

5-D



15-D



From: http://www.cs.reading.ac.uk/archive/hypercubes/

Non-CS uses of Hypercubes: chemistry

Welcome to
Hypercube, Inc.

Purchase a Product Online!

Products, Free Software and Demos

Developer Program

Employment at Hypercube

Contact Us

License your HyperChem 5 Demo

Experimental Compute Site

NEW

Laboratory Exercises Using HyperChem reviewed.
Read a review of Laboratory Exercises Using HyperChem in the August 1999 issue of the Journal of Chemical Education.

Purchase HyperChem Standard Online
For the first time, Hypercube, Inc. is offering a version of HyperChem Standard Release 5.11 that can be purchased electronically.

HyperChem 5.1 to 5.11 Update now available.
This update is for for Standard and Professional versions of HyperChem 5.1.

Latest News - December 1998, Pocket HyperChem for Windows CE
Pocket HyperChem for Windows CE and handheld PCs is now shipping!

What's Hot at Hypercube:

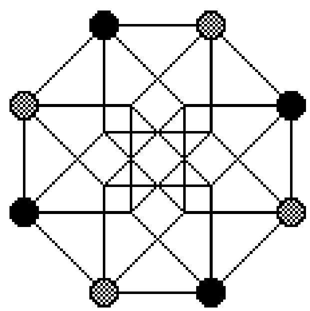
Hypercube Game

"Sudden Death in the 4th Dimension":

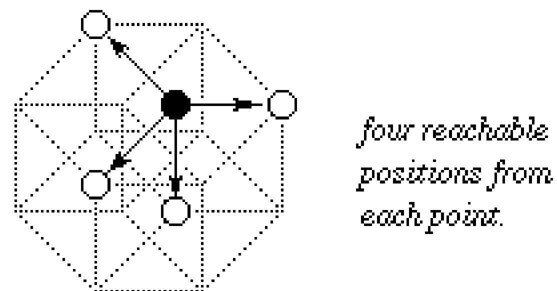
http://kbs.cs.tu-berlin.de/~jutta/swd/hyper-game.html

2 players,

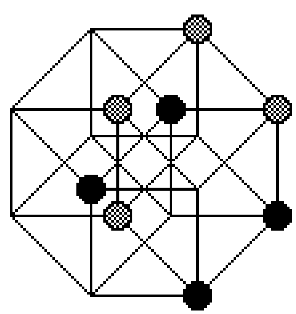
Start:



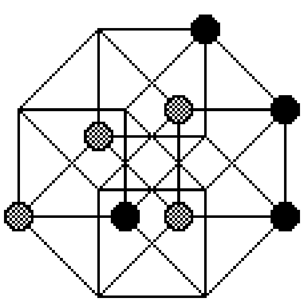
Move:



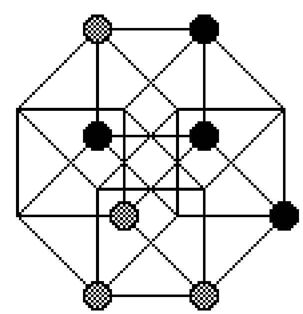
Win = all of your pieces in sequence or star



(Grey wins.)



(Grey wins.)



(Black wins.)

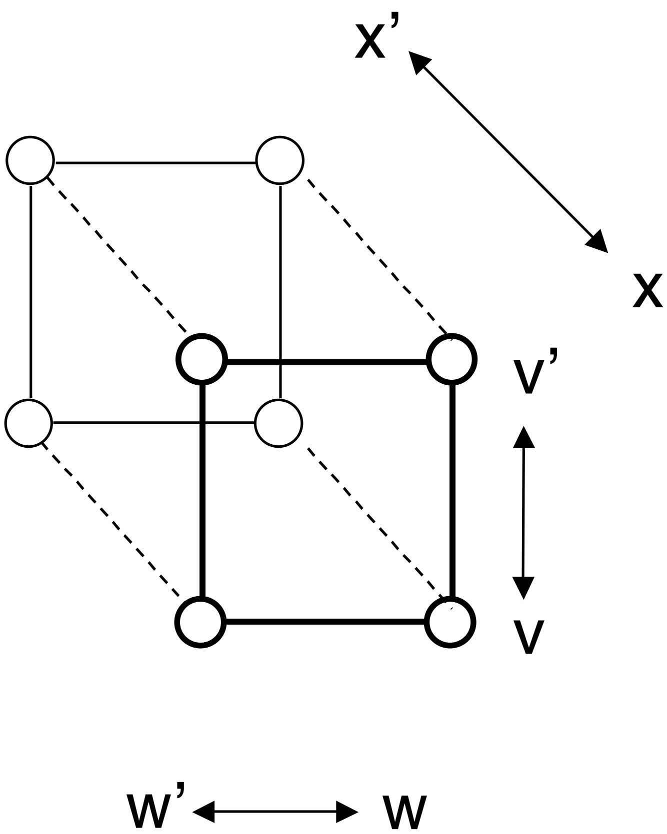
Rubick's Hypercube

http://www.rose-hulman.edu/~berglunb/Rubik/index.html

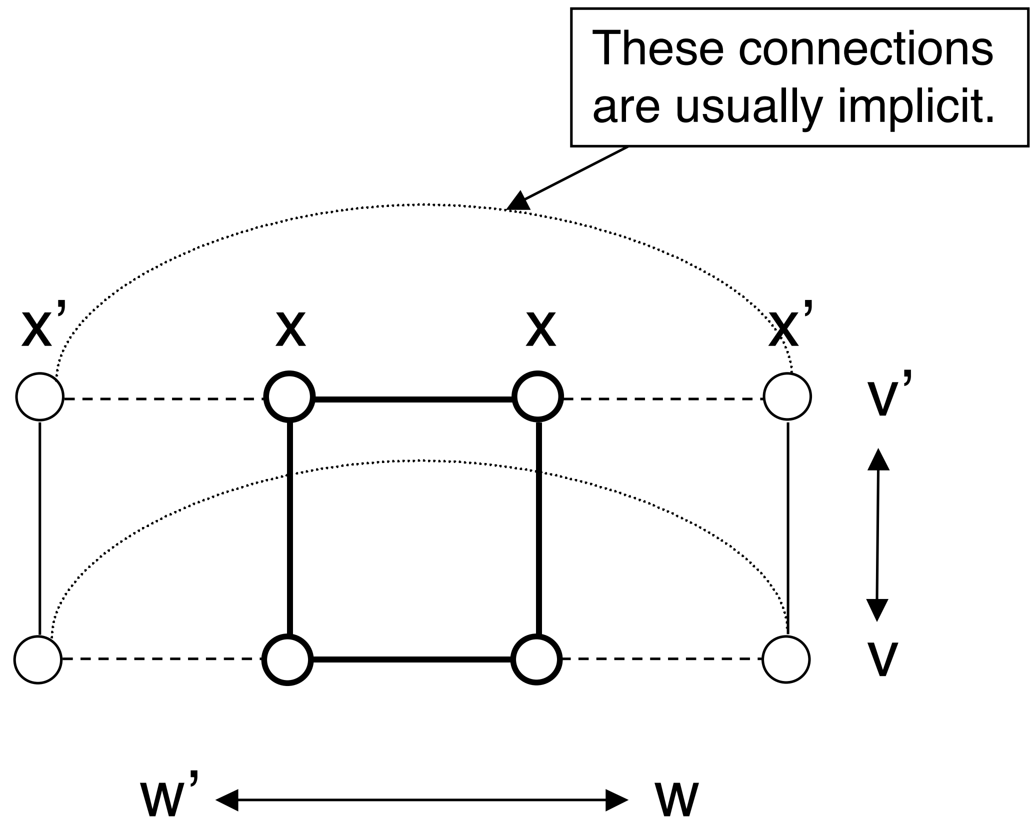
Karnaugh Maps

- Invented by Maurice Karnaugh, 1953, at Bell Labs
- A way to **visualize** hypercubes of up to 4 (and stretching to 5 or 6) dimensions
- Approach by “flattening” a hypercube
- A structured form of Venn Diagram

"3-D" Karnaugh Map

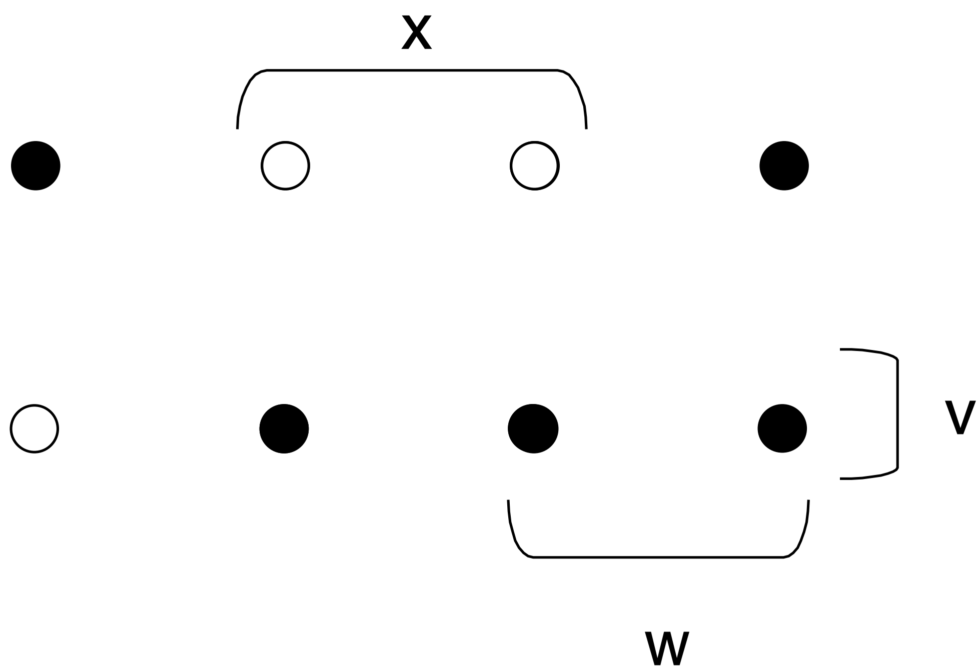


The Cube

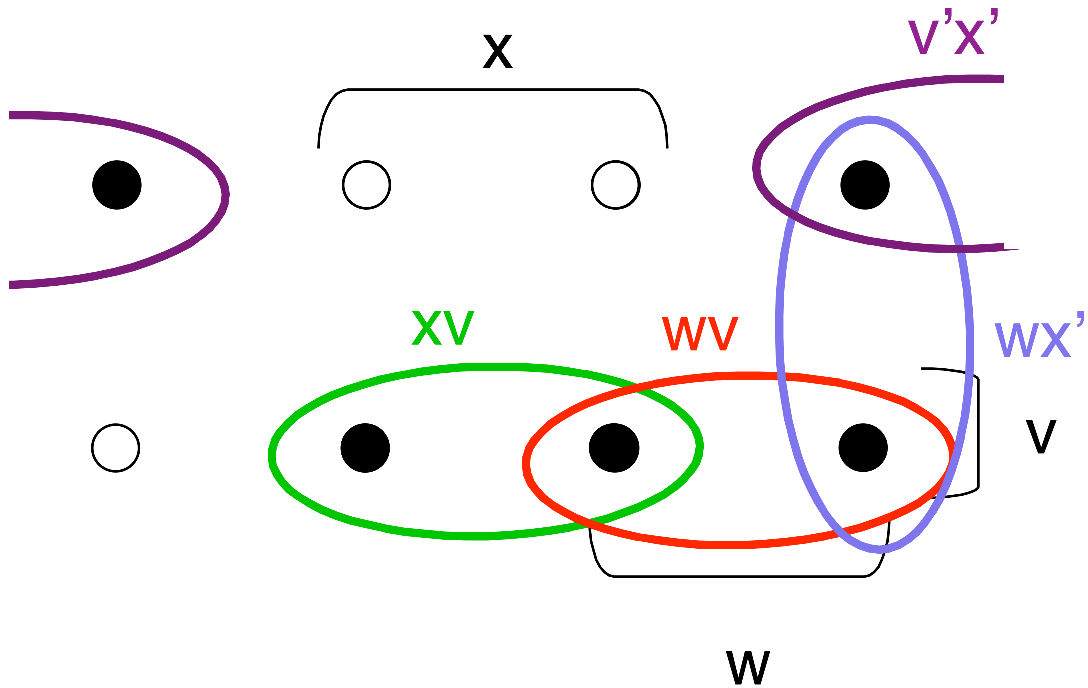


The Map

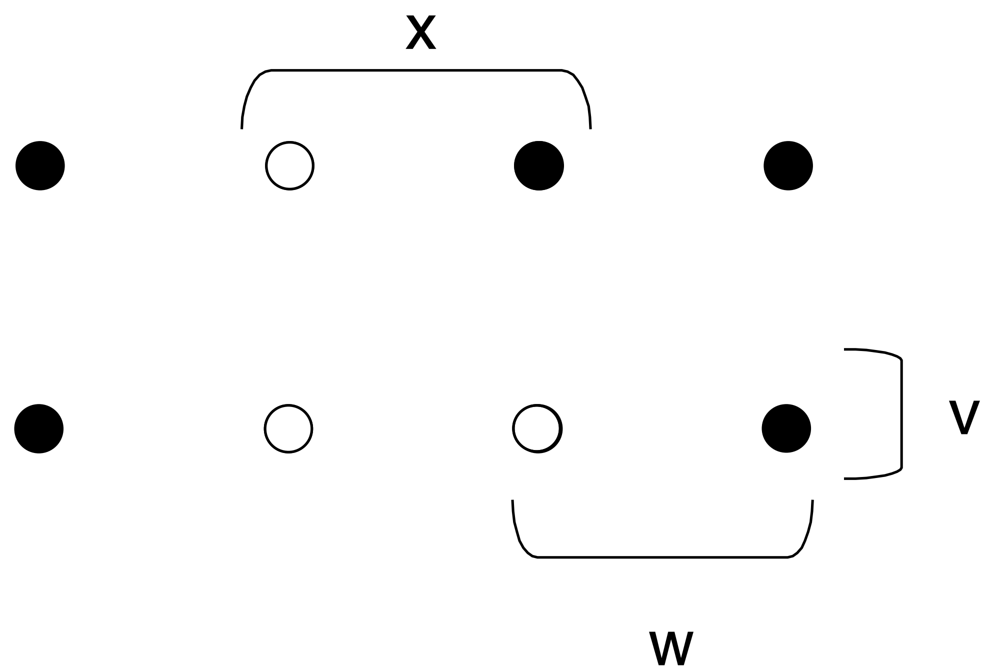
Covering on a Karnaugh Map



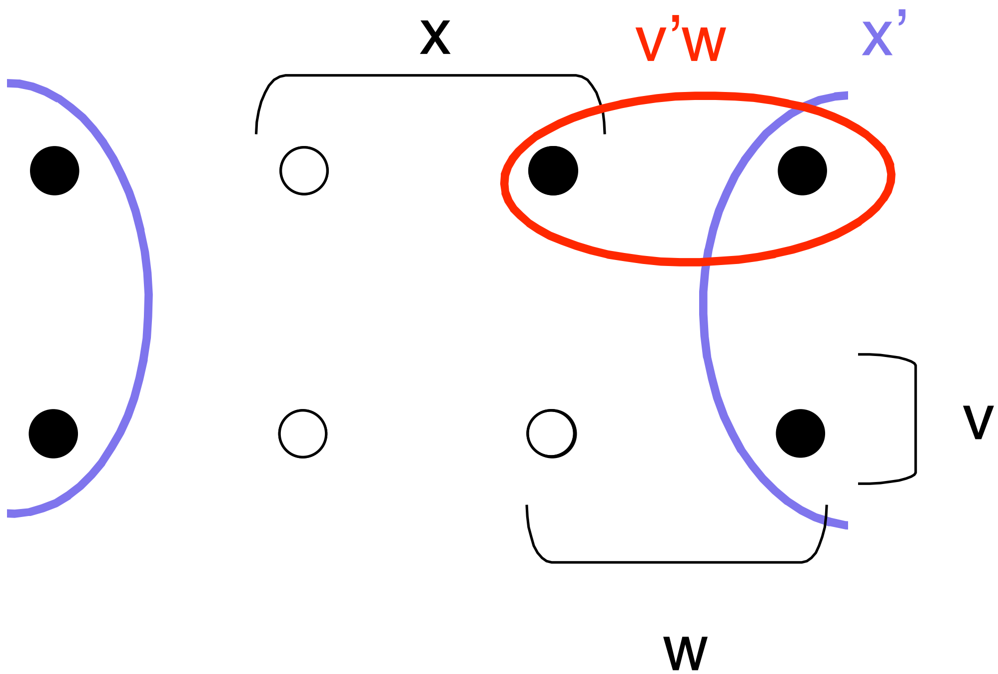
Covering on a Karnaugh Map



Try this one

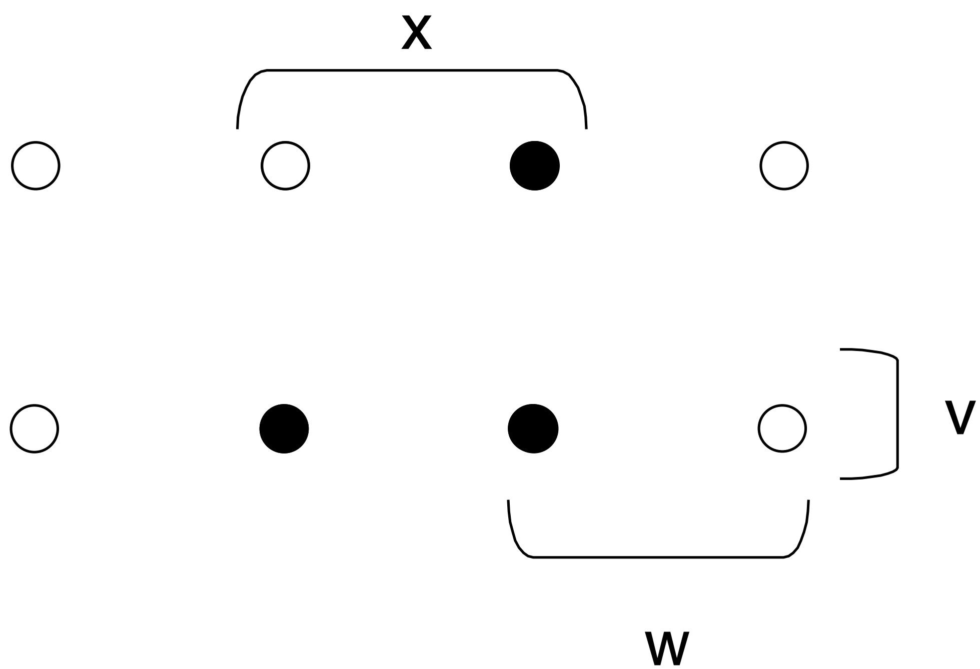


Answer



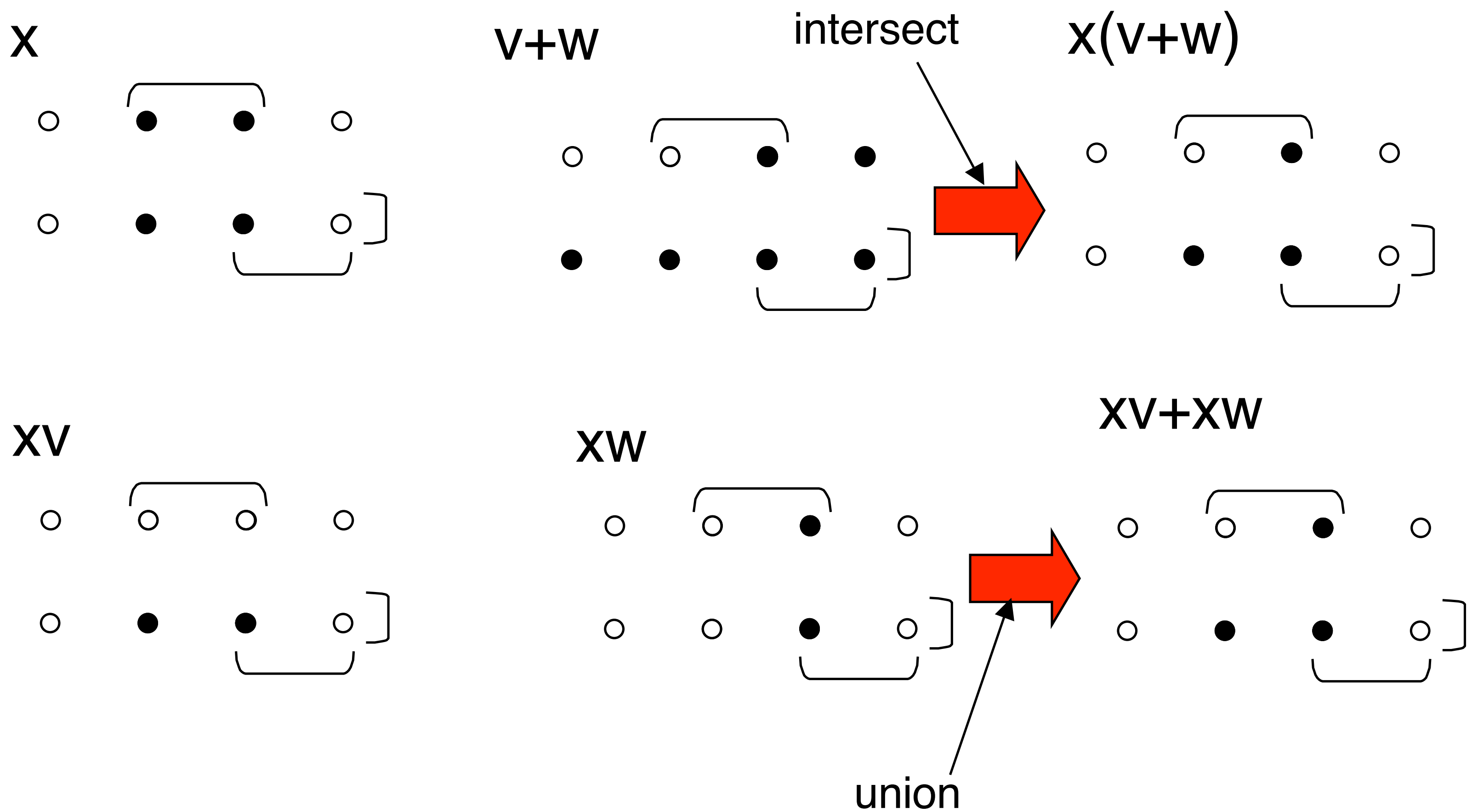
Using a Karnaugh Map to Prove Equalities

Prove that x(v+w) = xv + xw

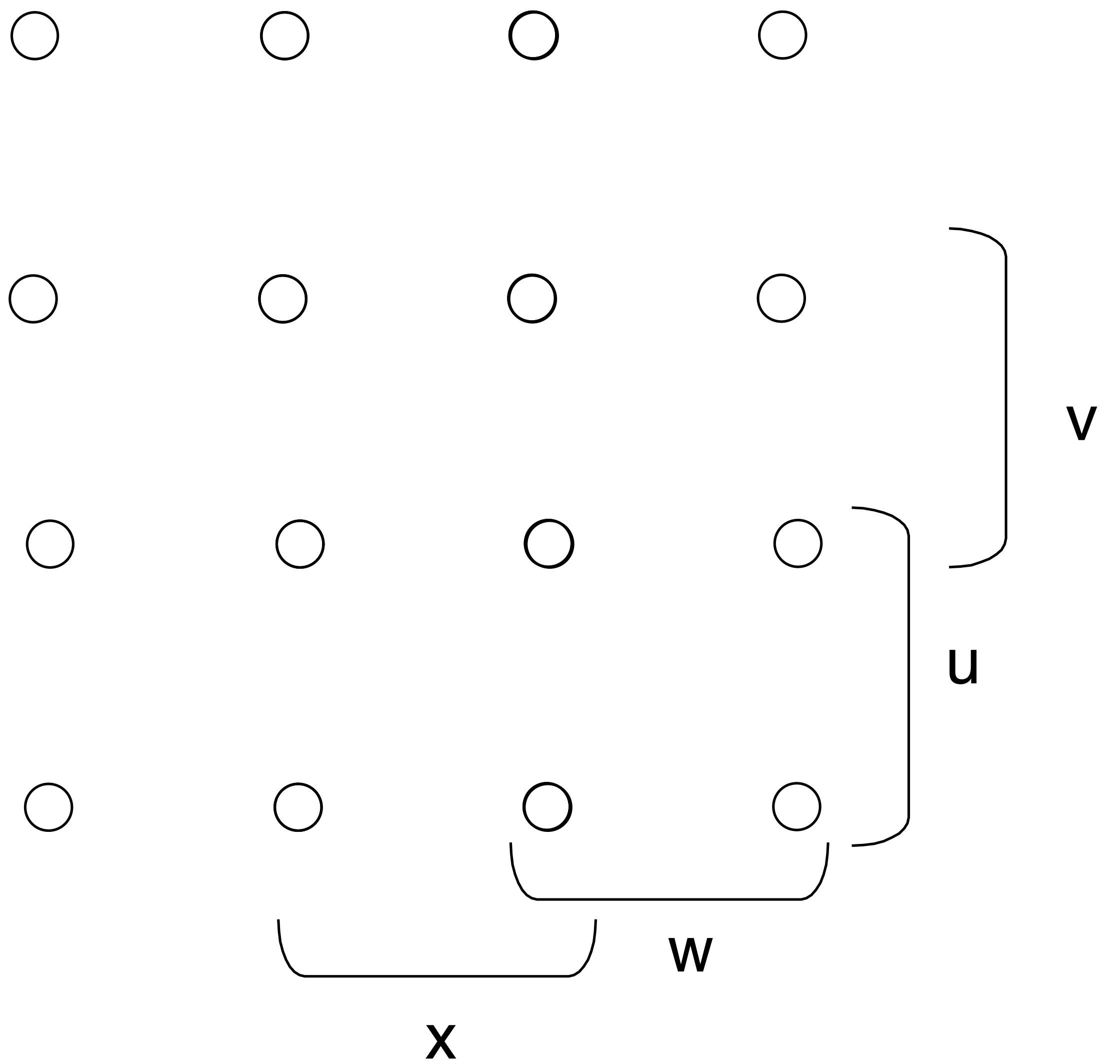


Using a Karnaugh Map to Prove Equalities

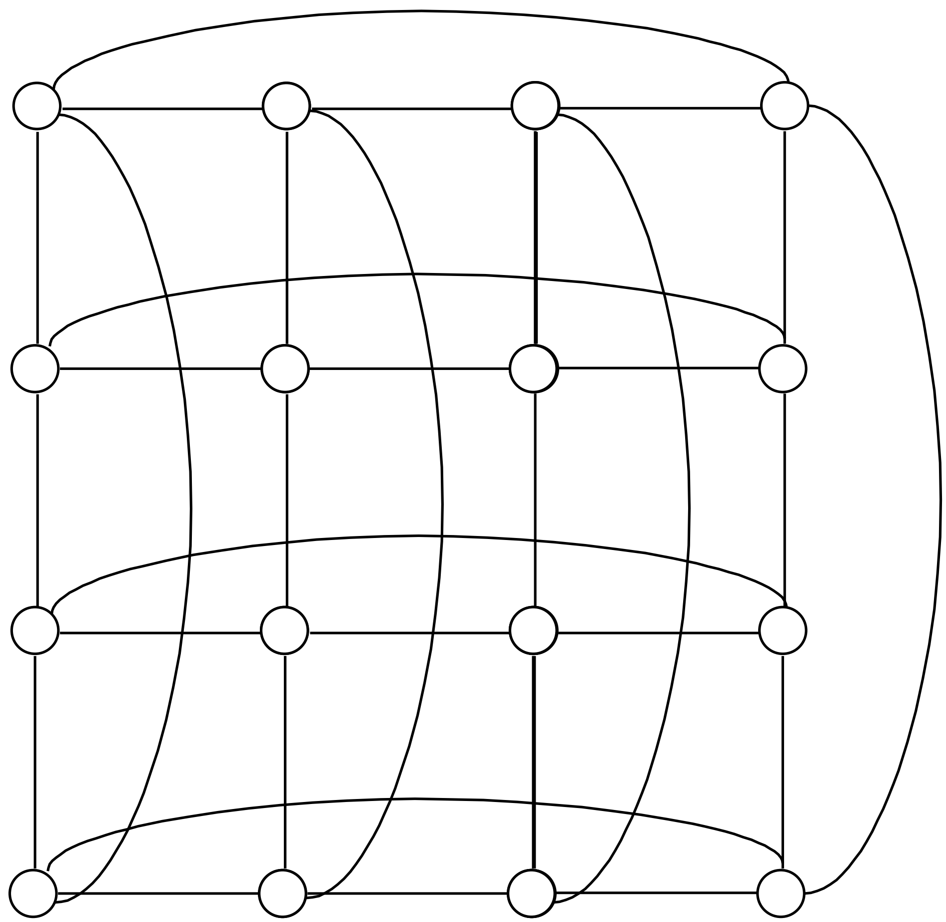
Prove that x(v+w) = xv + xw



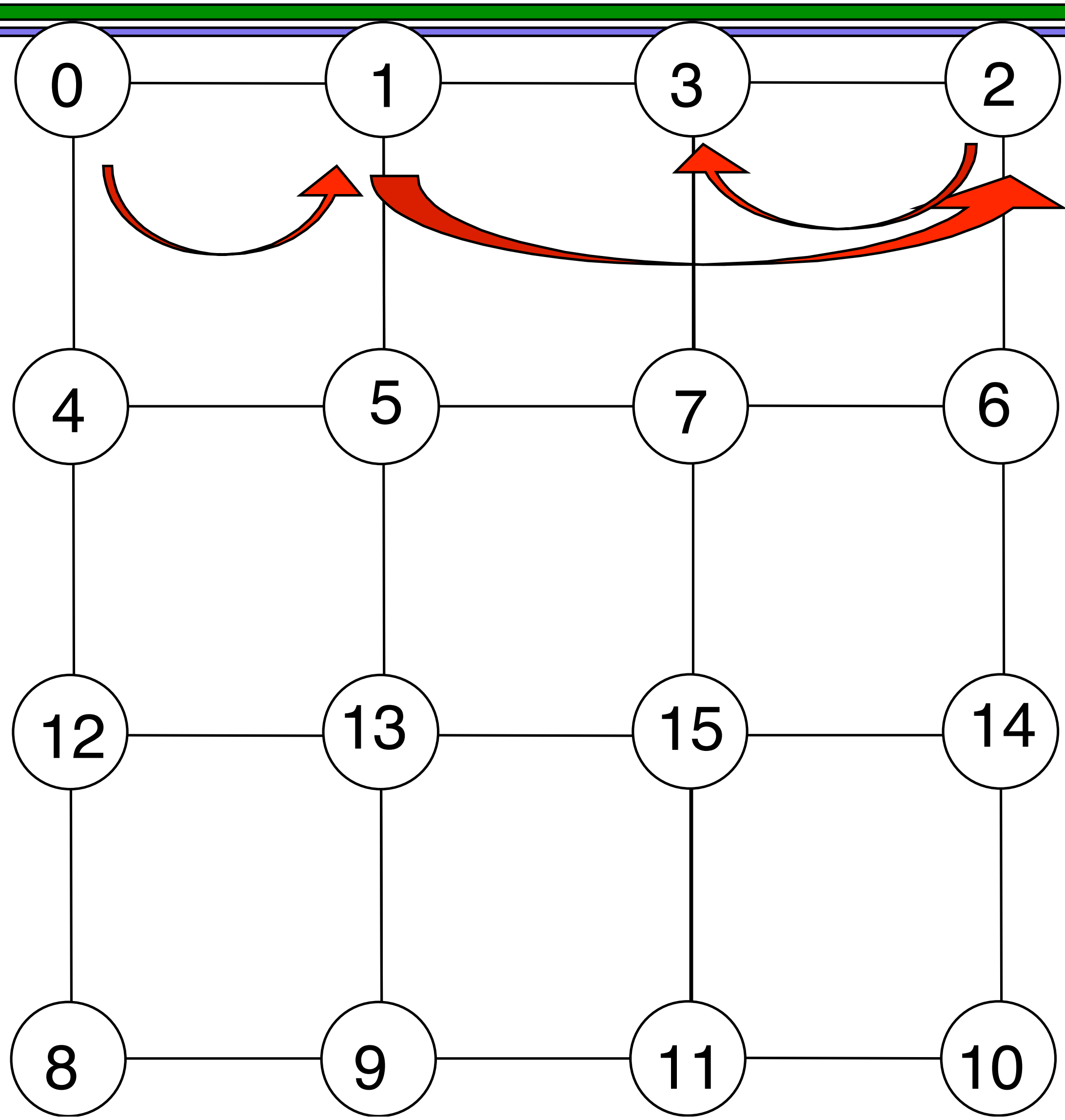
4-D Karnaugh Map



Implied connections on 4-D Karnaugh Map



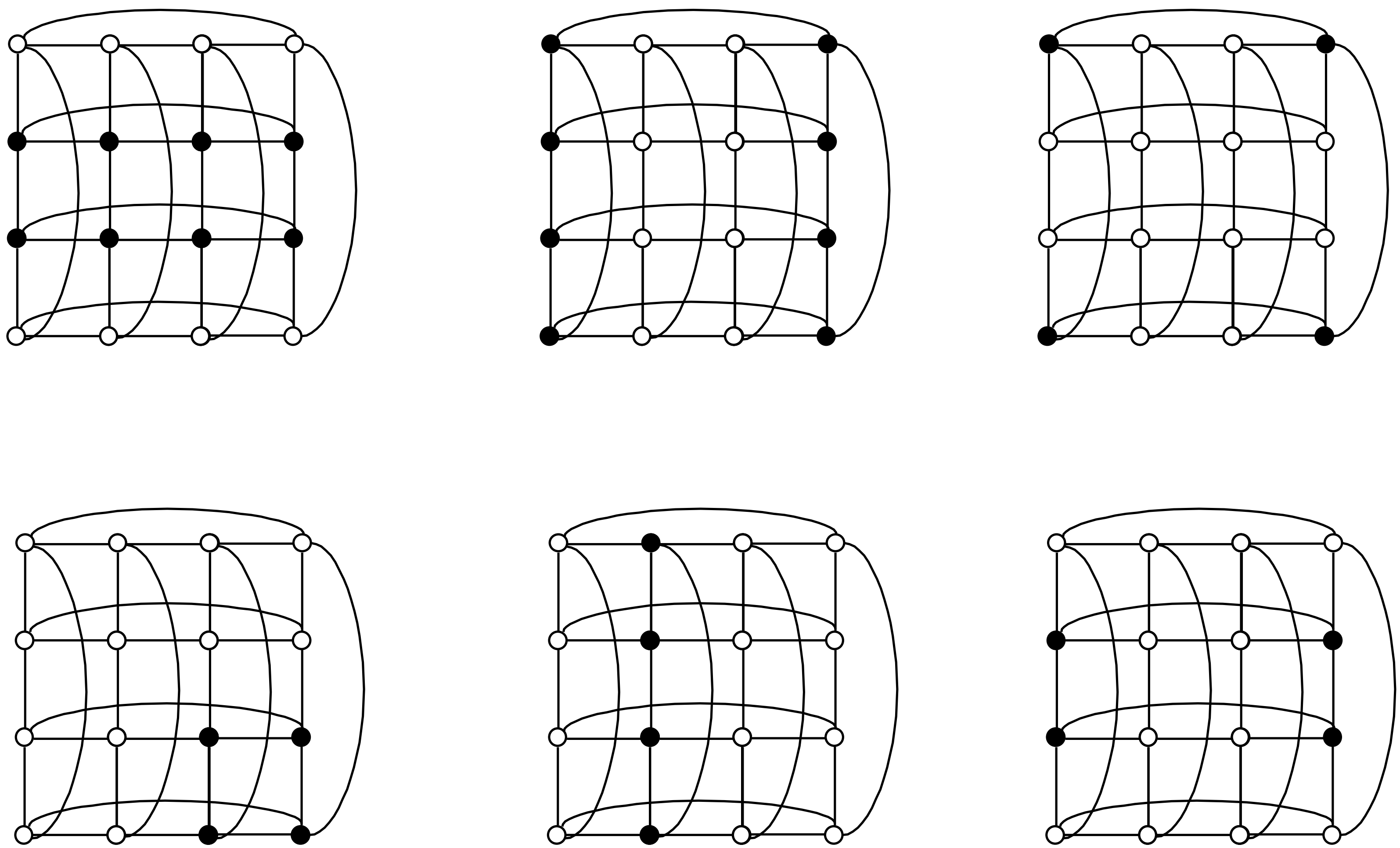
Minterm Numbering on 4-D Karnaugh Map



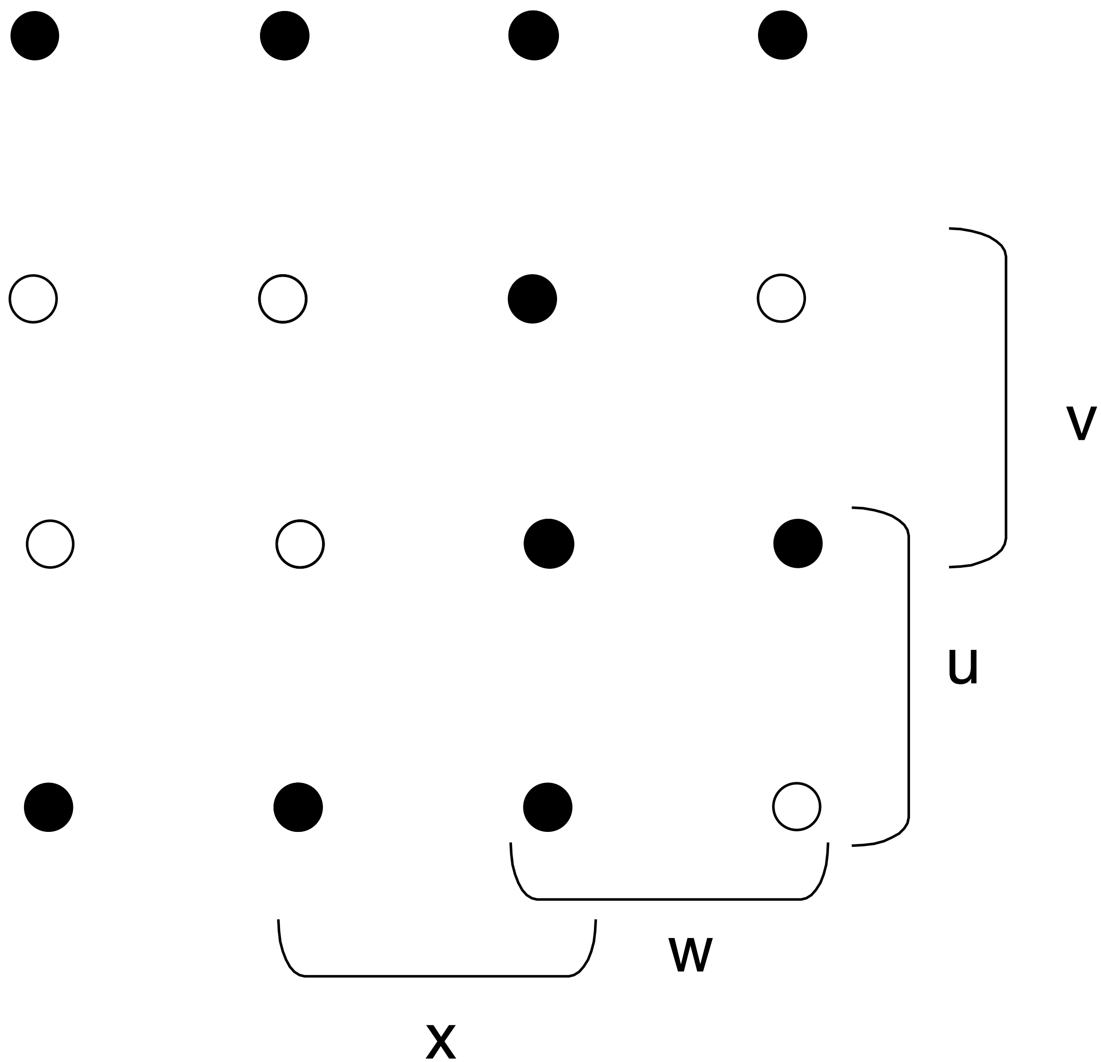
pattern, horizontal and vertical

Usage: Given a function in the form {1, 5, 7, 15}, for example

Assorted Sub-Cubes on 4-D Karnaugh Maps



Try this one



Exercises, etc.

- Project: Translate the hypercube game into a Karnaugh-map game.

- Exercise: Why is a Gray code sometimes called a “snake-in-the-box” code?

Simplification with "Don't Care" Combinations

- For certain problems, certain combinations (truth-table rows) never arise in actual operation.
- These are called "don't care" combinations.
- Because their input combinations never occur, the function value can be either 0 or 1. This means we can ***elect*** which value to use at our discretion.
- Usually we elect whichever value achieves the best simplification of the result.

Example: mod 3 adder

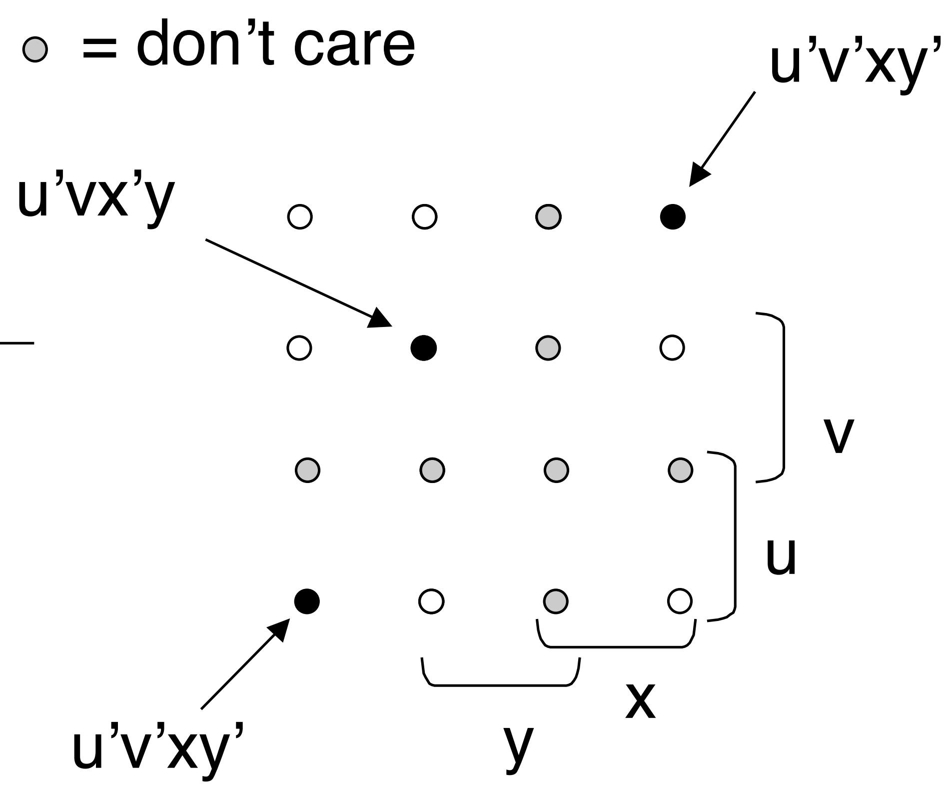
		uv		
		00	01	10
	00	0	0	1
xy	01	0	1	0
	10	1	0	0

		uv		
		00	01	10
	00	0	1	0
xy	01	1	0	0
	10	0	0	1

Only 9 of 16 combinations actually occur, leaving 7 don't care ones. The don't cares are the same for both functions.

Plot Function on a K-Map

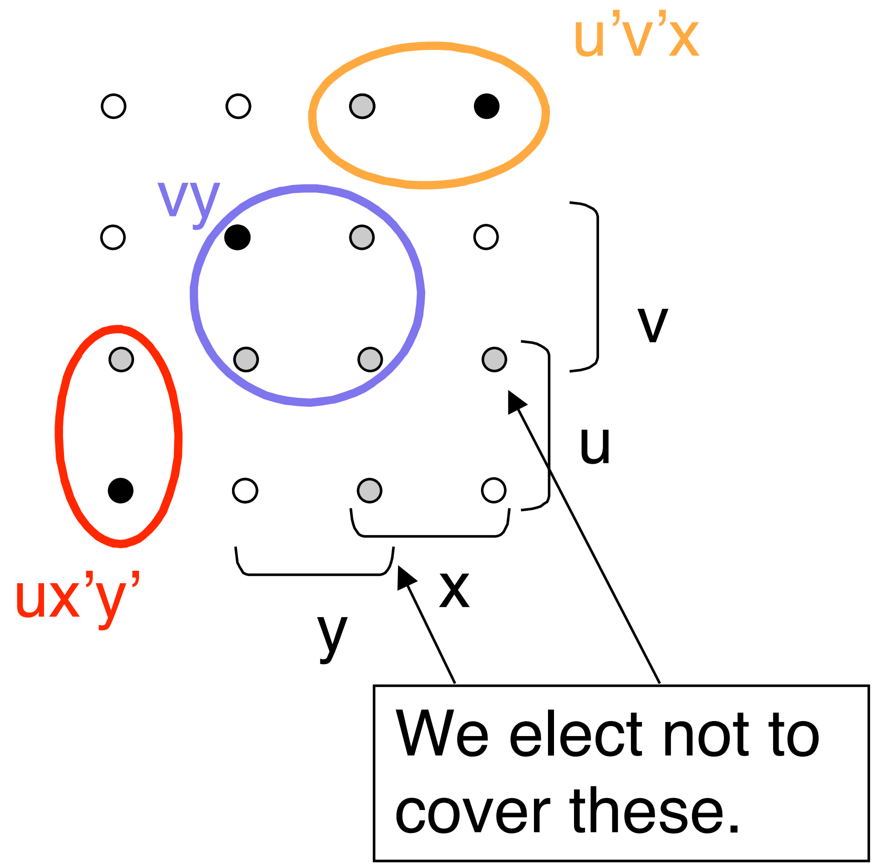
		uv		
		00	01	10
	00	0	0	1
xy	01	0	1	0
	10	1	0	0



Plot Function on a K-Map

○ = don’t care

		uv		
		00	01	10
xy	00	0	0	1
	01	0	1	0
	10	1	0	0



Final expression: vy + u’v’x + ux’y’

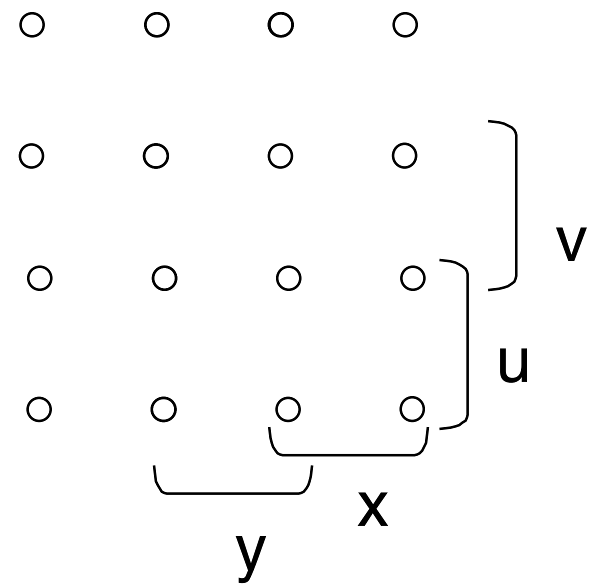
vs. original

u’ v' w x' + u' v w ' x + u v' w x'

Do the other half.

○ = don't care

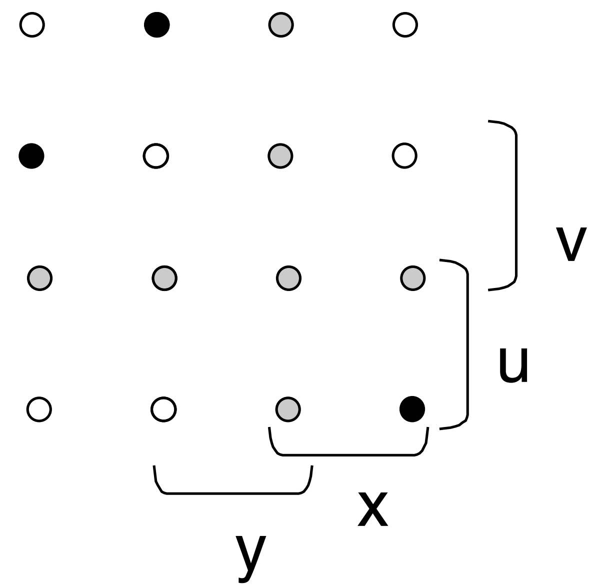
		uv		
		00	01	10
	00	0	1	0
xy	01	1	0	0
	10	0	0	1



The other half.

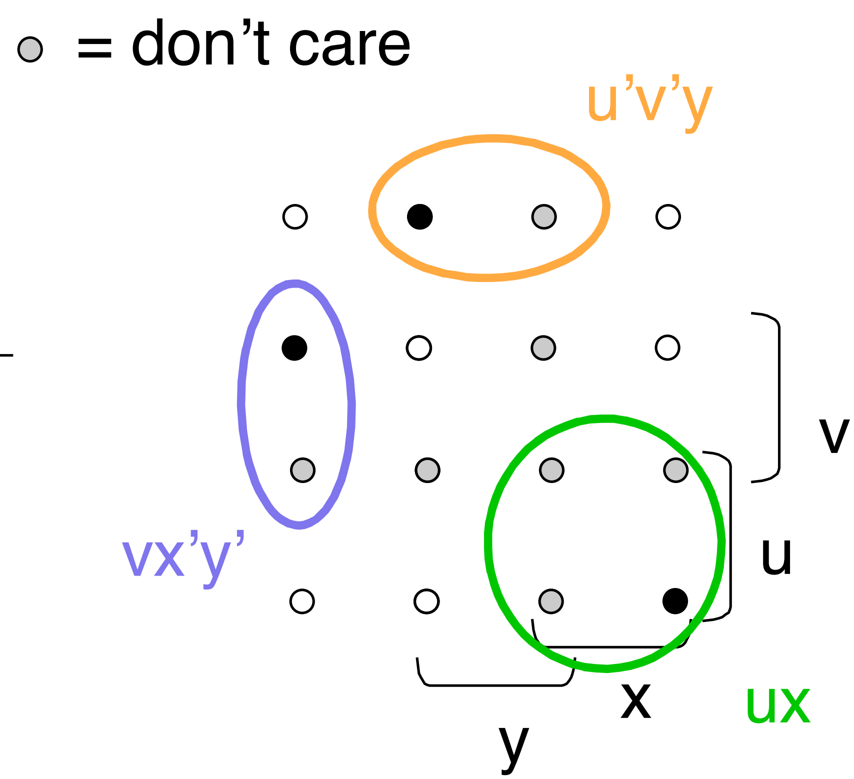
= don’t care

		uv 00	uv 01	uv 10
xy	00	0	1	0
xy	01	1	0	0
xy	10	0	0	1



The other half.

		uv		
		00	01	10
	00	0	1	0
xy	01	1	0	0
	10	0	0	1



Final expression: u'v'y + vx'y' + ux

vs. original

u' v' w' x + u' v w' x' + u v' w x'

Implementation using NAND gates

- In some families of logic, ***and***- and ***or***-gates might not be available.
- Instead the only gate is ***nand*** (negated *and*).
- The question of sufficiency arises.

Sufficiency

- We know that functions ***and***, ***or***, and ***not*** are **sufficient** to realize any logic function.
- Two proofs:
  - minterm expansion: a sum of minterms, and each minterm uses **and** and **not** only
  - Boole-Shannon expansion (applied recursively)

Sufficiency

- To show that a **given set of gate types** is sufficient, it is enough to show that ***and***, ***or***, and ***not*** can be realized with those gate types.

- Actually it is sufficient to show just ***and*** and ***not*** *can be realized*, since ***or*** can be realized as:

  $$a + b = (a' b')'$$

  by DeMorgan's law. So if ***and*** and ***not*** are realizable, so is ***or***.

nand alone is sufficient

- $a' = \mathrm{nand}(a, a)$
- $\mathrm{and}(a, b) = \mathrm{nand}(a, b)'$
- $\mathrm{or}(a, b) = \mathrm{nand}(a', b')$

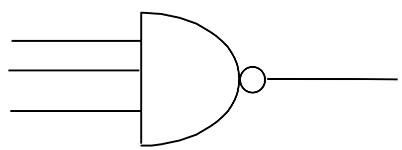
  $= \mathrm{nand}(\mathrm{nand}(a, a), \mathrm{nand}(b, b))$

Exercises

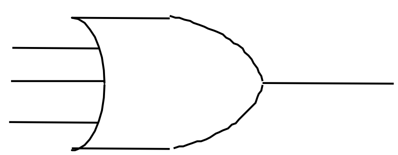
- ***nor*** alone is sufficient.
- Is ***xor*** alone sufficient?

Bubble Logic

nand



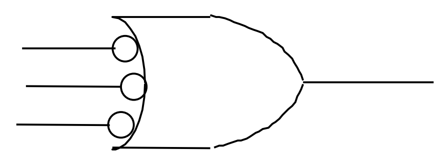
or

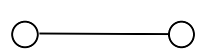


DeMorgan:

nand(a, b) =
and(a, b)’ =
or(a’, b’)

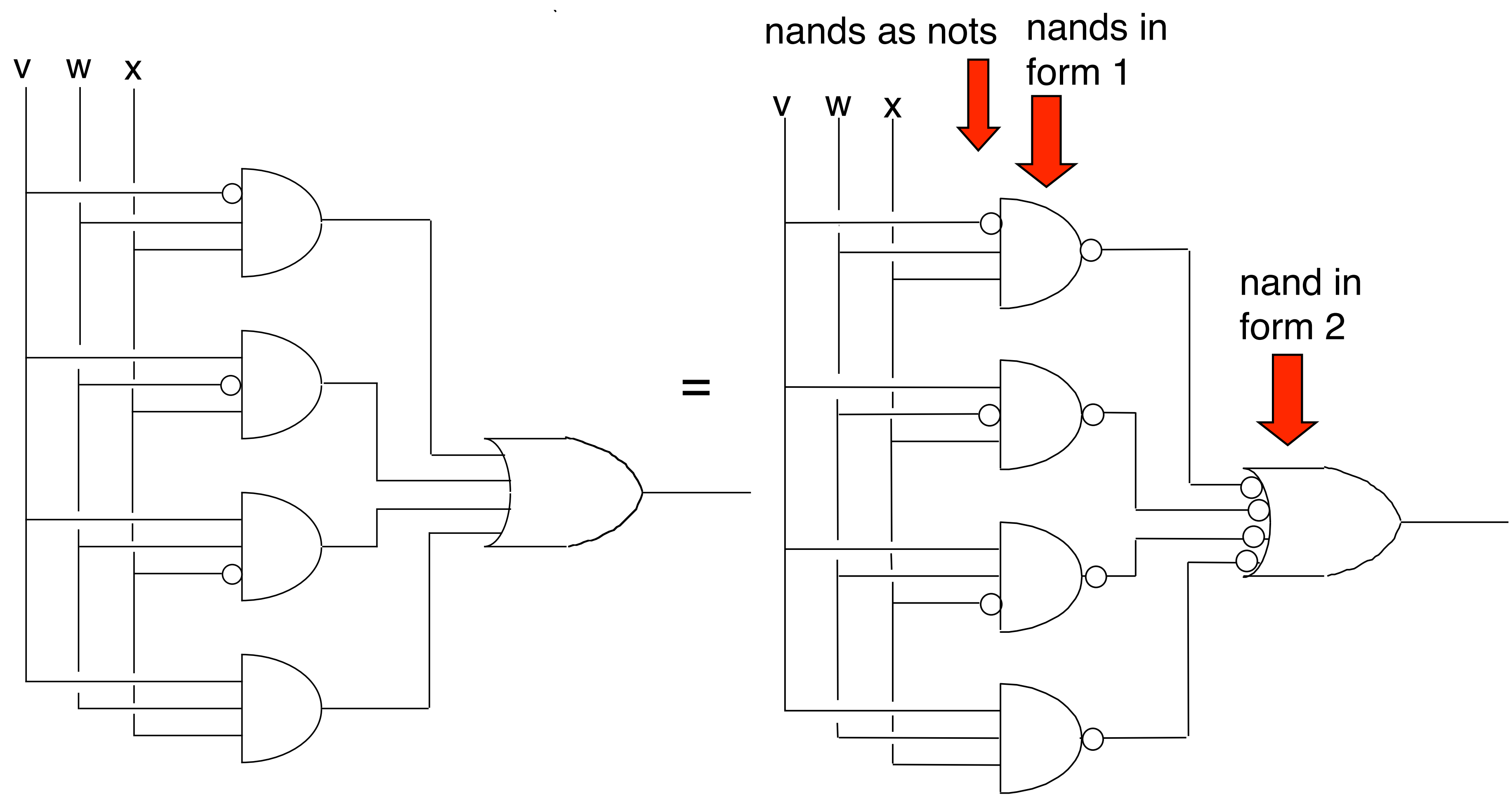
nand, another way



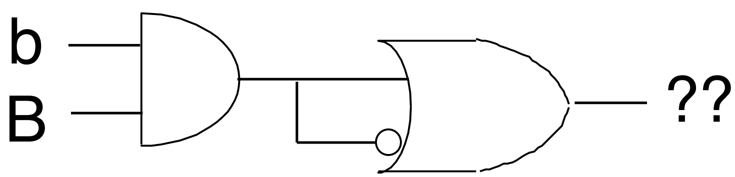
 =

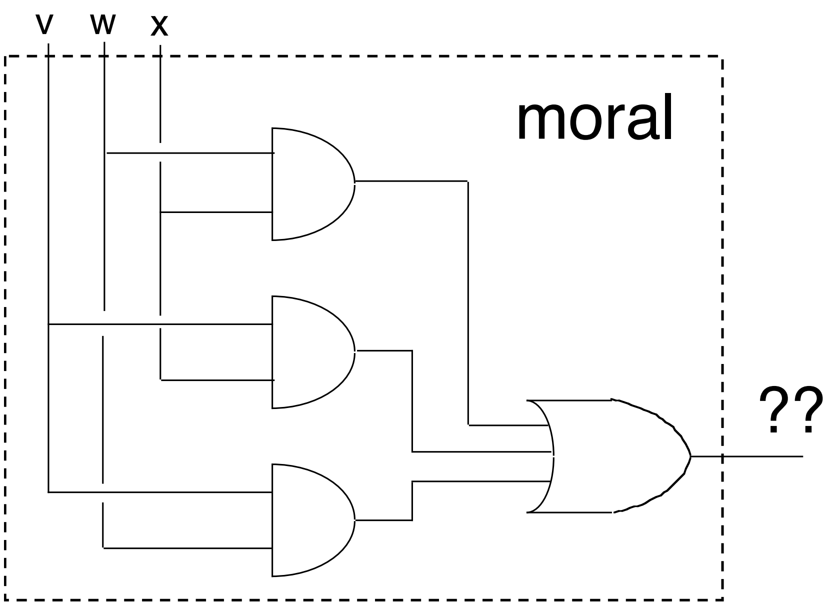
(a’)’ = a

SOP form using nand's only



Logic Rebuses

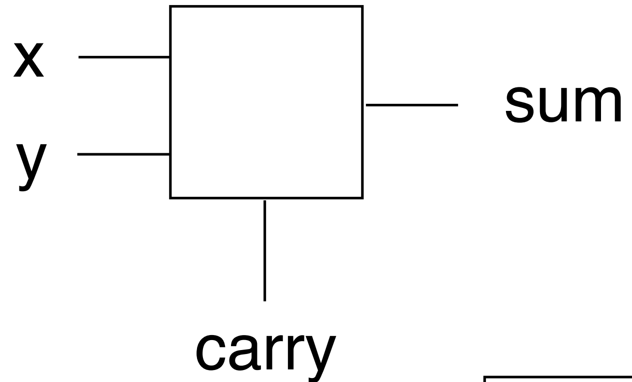




Common Logic Packages

- mod-2 adder
- mod-2 adder with carry-in
- 4-bit adder
- decoder (binary to one-hot)
- encoder (one-hot to binary)
- (MUX) multiplexor
- (DMUX) demultiplexor

mod-2 adder with carry-out

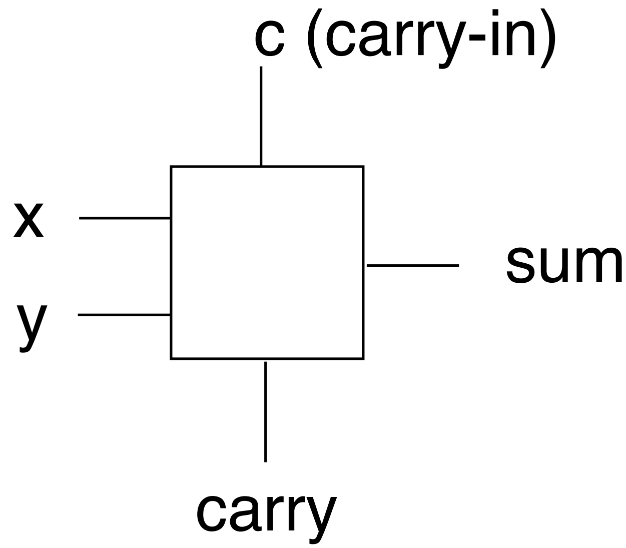


exclusive or

$$\text{sum}(x, y) = x \oplus y = xy' + x'y$$

$$\text{carry}(x, y) = x\,y$$

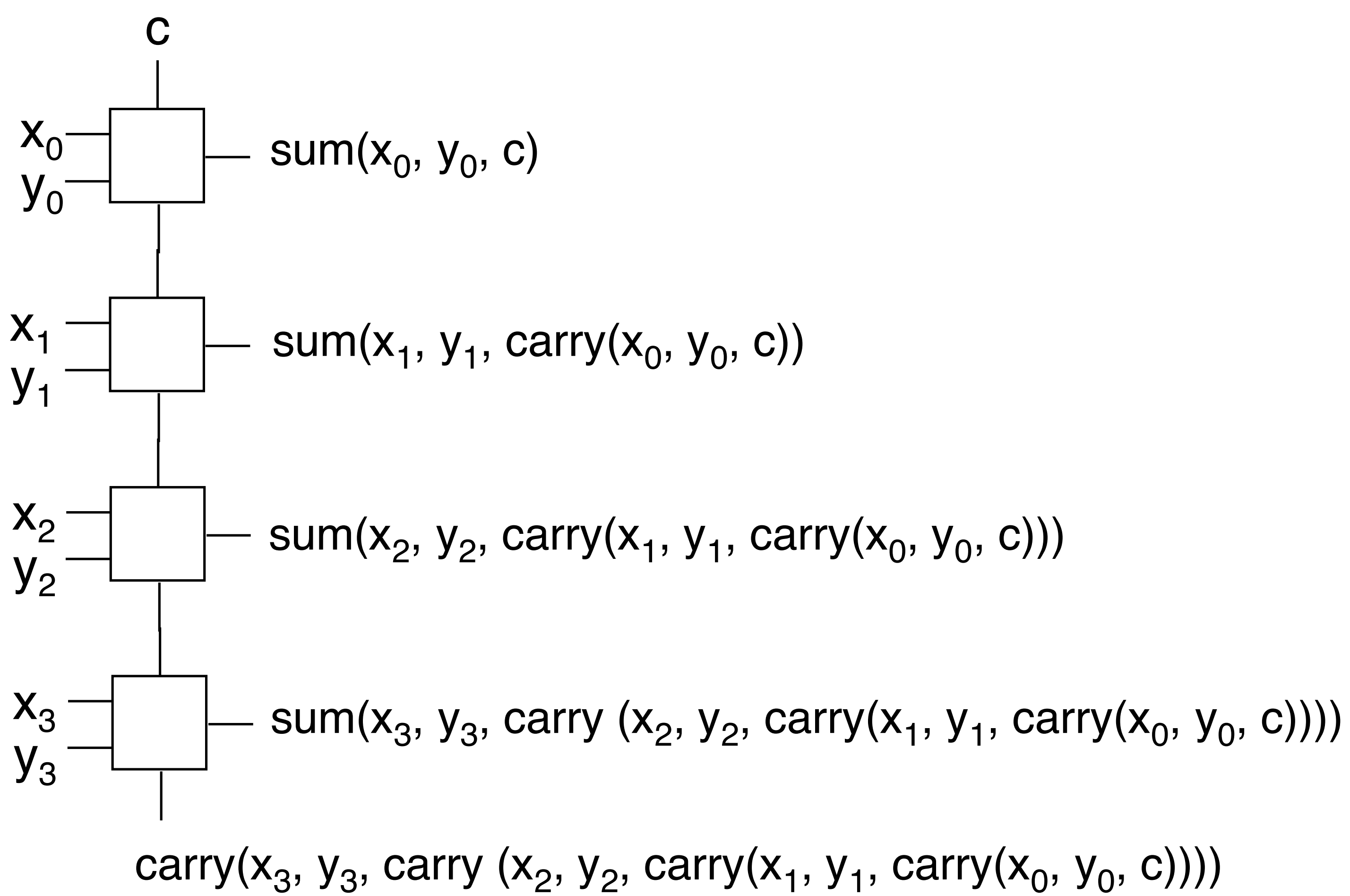
mod-2 adder with carry-in/out



$$\text{sum}(x, y, c) = x \oplus y \oplus c$$

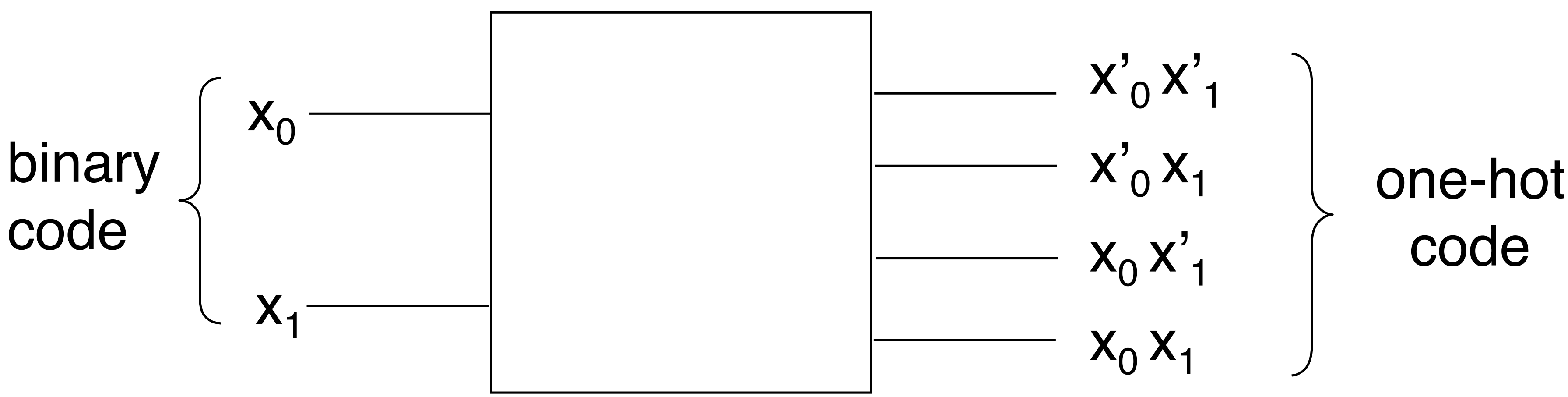
$$\text{carry}(x, y, c) = x\,y + x\,c + y\,c = \text{majority}(x, y, c)$$

4-bit ripple-carry adder

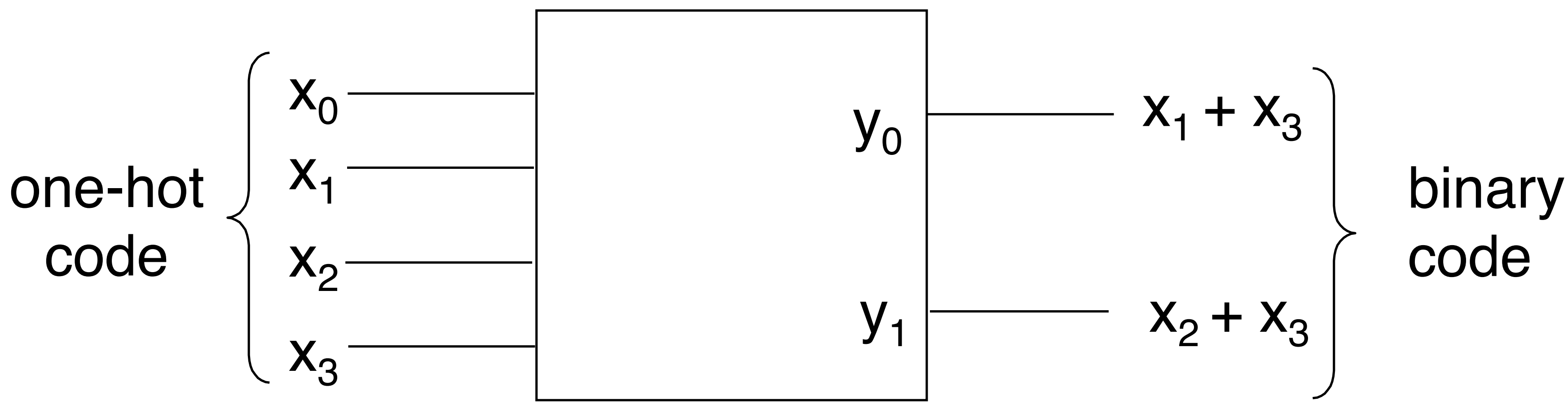


Note: There are other ways to implement this.

4-bit decoder



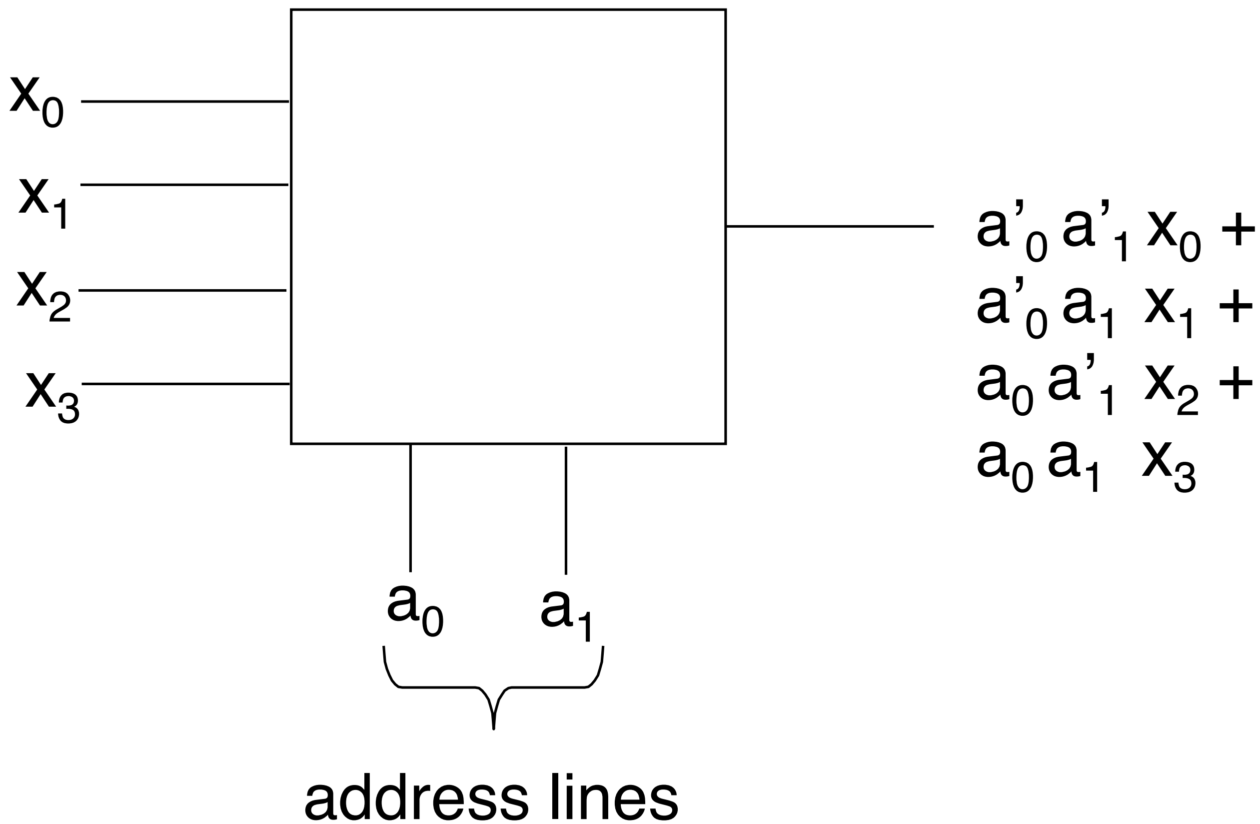
4-bit encoder



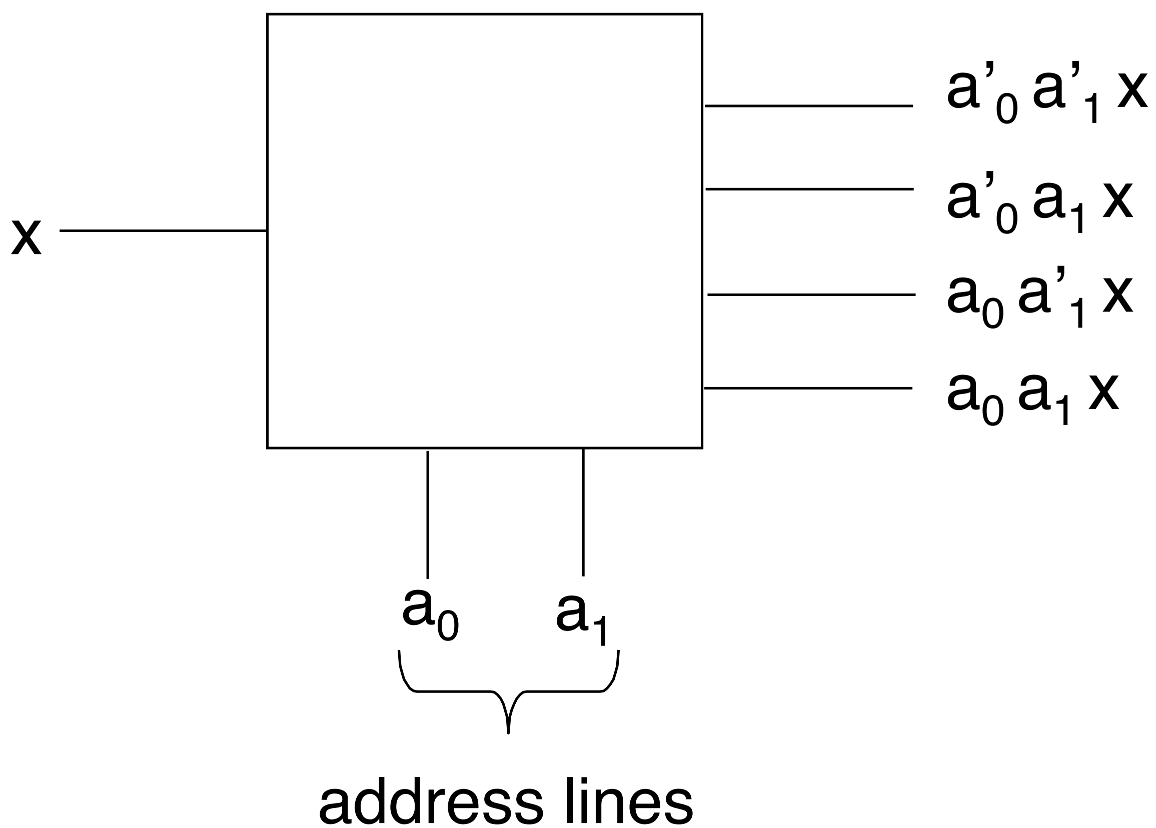
unlisted combinations are assumed not to occur

x_0	x_1	x_2	x_3	y_1	y_0
1	0	0	0	0	0
0	1	0	0	0	1
0	0	1	0	1	0
0	0	0	1	1	1

multiplexor



demultiplexor



Exercises

- How would you build a decoder out of a demultiplexor?
- How would you build a 16-way multiplexor out of 4-way multiplexors?
- How would you build a 16-way demultiplexor out of 4-way demultiplexors?
- How would you use a decoder to build a decoder ring?